Contents

Preface ......................................................................................................................................................... xxxi
  Audience............................................................................................................................................... xxxi
  Documentation Accessibility .................................................................................................................. xxxi
  Related Documents ............................................................................................................................... xxxii
  Conventions .......................................................................................................................................... xxxii
  Syntax Descriptions ............................................................................................................................... xxxiii

What’s New in PL/SQL? ........................................................................................................................... xxxv
  New PL/SQL Features for 11g Release 1 (11.1) ................................................................................... xxxv

1 Overview of PL/SQL
  Advantages of PL/SQL .......................................................................................................................... 1-1
    Tight Integration with SQL .................................................................................................................. 1-1
    High Performance .............................................................................................................................. 1-2
    High Productivity .............................................................................................................................. 1-3
    Full Portability .................................................................................................................................. 1-3
    Tight Security ..................................................................................................................................... 1-3
    Access to Predefined Packages ......................................................................................................... 1-3
    Support for Object-Oriented Programming ....................................................................................... 1-3
    Support for Developing Web Applications and Server Pages ......................................................... 1-4
  Main Features of PL/SQL ....................................................................................................................... 1-4
    PL/SQL Blocks ................................................................................................................................ 1-4
    PL/SQL Error Handling ...................................................................................................................... 1-5
    PL/SQL Input and Output ..................................................................................................................... 1-6
    PL/SQL Variables and Constants ......................................................................................................... 1-6
      Declaring PL/SQL Variables ............................................................................................................ 1-7
      Assigning Values to Variables ........................................................................................................... 1-7
      Declaring PL/SQL Constants ............................................................................................................ 1-9
      Bind Variables ................................................................................................................................. 1-9
    PL/SQL Data Abstraction ..................................................................................................................... 1-9
      Cursors ........................................................................................................................................ 1-10
      %TYPE Attribute ............................................................................................................................ 1-10
      %ROWTYPE Attribute ...................................................................................................................... 1-10
      Collections ..................................................................................................................................... 1-11
      Records .......................................................................................................................................... 1-12
2 PL/SQL Language Fundamentals

Character Sets and Lexical Units .................................................................................. 2-1
  Delimiters ..................................................................................................................... 2-2
  Identifiers ..................................................................................................................... 2-4
    Reserved Words and Keywords .................................................................................. 2-5
    Predefined Identifiers ............................................................................................... 2-5
    Quoted Identifiers ..................................................................................................... 2-5
  Literals ............................................................................................................................ 2-6
    Numeric Literals .......................................................................................................... 2-6
    Character Literals ....................................................................................................... 2-7
    String Literals ............................................................................................................. 2-7
    BOOLEAN Literals ....................................................................................................... 2-8
    Date and Time Literals .............................................................................................. 2-8
  Comments ...................................................................................................................... 2-9
    Single-Line Comments ............................................................................................... 2-9
    Multiline Comments ................................................................................................. 2-10
Declarations ................................................................................................................... 2-10
  Variables ...................................................................................................................... 2-11
  Constants ...................................................................................................................... 2-11
  Using DEFAULT .......................................................................................................... 2-11
  Using NOT NULL ......................................................................................................... 2-12
  Using the %TYPE Attribute ...................................................................................... 2-12
  Using the %ROWTYPE Attribute .............................................................................. 2-15
    Aggregate Assignment .............................................................................................. 2-16
    Using Aliases ............................................................................................................ 2-17
  Restrictions on Declarations ..................................................................................... 2-18
Naming Conventions .................................................................................................... 2-19
  Scope ............................................................................................................................ 2-19
  Case Sensitivity .......................................................................................................... 2-20
  Name Resolution ........................................................................................................ 2-20
  Synonyms .................................................................................................................... 2-22
Scope and Visibility of PL/SQL Identifiers .................................................................. 2-22
## 6 Using Static SQL

### Using Multidimensional Collections

- Managing Cursors in PL/SQL ................................................................. 6-7

### Using Collection Methods

- SQL Cursors (Implicit) ........................................................................... 6-7
- Attributes of SQL Cursors .................................................................... 6-8
  - %FOUND Attribute: Has a DML Statement Changed Rows? ............... 6-8
  - %ISOPEN Attribute: Always FALSE for SQL 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 SQL Cursors ................................. 6-9

### Using Records as Subprogram Parameters and Function Return Values

- Explicit Cursors ....................................................................................... 6-9
- Declaring a Cursor ................................................................................... 6-10
- Opening a Cursor ..................................................................................... 6-11
- Fetching with a Cursor ............................................................................ 6-11

### Assigning Values to Records

- Comparing Records ................................................................................ 5-34
- Inserting Records Into the Database ...................................................... 5-36
- Updating the Database with Record Values ......................................... 5-37
- Restrictions on Record Inserts and Updates ........................................ 5-38
- Querying Data Into Collections of Records ........................................ 5-38

### Avoiding Collection Exceptions

- Checking If a Collection Element Exists (EXISTS Method) .................. 5-21
- Checking the Maximum Size of a Collection (LIMIT Method) .......... 5-21
- Finding the First or Last Collection Element (FIRST and LAST Methods) 5-23
- Looping Through Collection Elements (PRIOR and NEXT Methods) .... 5-24
- 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

### Defining and Declaring Records

- Using Collection Methods ..................................................................... 5-20
- Using Records as Subprogram Parameters and Function Return Values 5-33
- Assigning Values to Records ................................................................. 5-34

### Using Static SQL

- Using Multidimensional Collections .................................................. 5-19
- Using Collection Methods ..................................................................... 5-20
- Using Records as Subprogram Parameters and Function Return Values 5-33
- Assigning Values to Records ................................................................. 5-34

### Description of Static SQL

- Data Manipulation Language (DML) Statements .................................. 6-1
- Transaction Control Language (TCL) Statements .................................. 6-3
- SQL Functions ......................................................................................... 6-3
- SQL Pseudocolumns .............................................................................. 6-4
  - CURRVAL and NEXTVAL ................................................................. 6-4
  - LEVEL ......................................................................................... 6-5
  - ROWID ...................................................................................... 6-5
  - ROWNUM ................................................................................. 6-6
- SQL Operators ........................................................................................ 6-6
  - Comparison Operators ...................................................................... 6-6
  - Set Operators ................................................................................. 6-7
  - Row Operators ............................................................................... 6-7
- Managing Cursors in PL/SQL ................................................................. 6-7
- SQL Cursors (Implicit) ........................................................................... 6-7
- Attributes of SQL Cursors .................................................................... 6-8
  - %FOUND Attribute: Has a DML Statement Changed Rows? ............... 6-8
  - %ISOPEN Attribute: Always FALSE for SQL 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 SQL Cursors ................................. 6-9
- Explicit Cursors ....................................................................................... 6-9
- Declaring a Cursor ................................................................................... 6-10
- Opening a Cursor ..................................................................................... 6-11
- Fetching with a Cursor ............................................................................ 6-11
### 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)

8-18

#### Choosing Between AUTHID CURRENT_USER and AUTHID DEFINER

8-19

#### AUTHID and the SQL Command SET ROLE

8-20

#### Need for Template Objects in IR Subprograms

8-20

#### Overriding Default Name Resolution in IR Subprograms

8-20

#### Using Views and Database Triggers with IR Subprograms

8-20

#### Using Database Links with IR Subprograms

8-21

#### Using Object Types with IR Subprograms

8-21

#### Invoking IR Instance Methods

8-22

### Using Recursive PL/SQL Subprograms

8-23

### Invoking External Subprograms

8-23

### Controlling Side Effects of PL/SQL Subprograms

8-25

### Understanding PL/SQL Subprogram Parameter Aliasing

8-25

### Using the PL/SQL Function Result Cache

8-27

#### Enabling Result-Caching for a Function

8-28

#### Developing Applications with Result-Cached Functions

8-29

#### Restrictions on Result-Cached Functions

8-29

#### Examples of Result-Cached Functions

8-30

##### Result-Cached Application Configuration Parameters

8-30

##### Result-Cached Recursive Function

8-32

### Advanced Result-Cached Function Topics

8-32

#### Rules for a Cache Hit

8-32

#### Bypassing the Result Cache

8-33

#### Making Result-Cached Functions Handle Session-Specific Settings

8-33

#### Making Result-Cached Functions Handle Session-Specific Application Contexts

8-34

#### Choosing Result-Caching Granularity

8-35

#### Result Caches in Oracle RAC Environment

8-36

#### Managing the Result Cache

8-37

#### Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend

8-38

## 9 Using Triggers

### Overview of Triggers

9-1

#### Trigger Types

9-2

#### Trigger States

9-2

#### Data Access for Triggers

9-2

#### Uses of Triggers

9-3

### Guidelines for Designing Triggers

9-3

### Privileges Required to Use Triggers

9-4

### Creating Triggers

9-5

#### Naming Triggers

9-6

#### When Does the Trigger Fire?

9-6

##### Do Import and SQL*Loader Fire Triggers?

9-6

##### How Column Lists Affect UPDATE Triggers

9-7

#### Controlling When a Trigger Fires (BEFORE and AFTER Options)

9-7
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 ......................... 10-3
Referencing PL/SQL Package Contents ......................................... 10-4
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
Predefined PL/SQL Exceptions ....................................................... 11-4
Defining Your Own PL/SQL Exceptions ......................................... 11-6
  Declaring PL/SQL Exceptions ................................................. 11-6
12 Tuning PL/SQL Applications for Performance

How PL/SQL Optimizes Your Programs ................................................................. 12-1
When to Tune PL/SQL Code ................................................................................. 12-2
Guidelines for Avoiding PL/SQL Performance Problems ................................     12-3
   Avoiding CPU Overhead in PL/SQL Code ........................................................ 12-3
      Make SQL Statements as Efficient as Possible ............................................. 12-3
      Make Function Calls as Efficient as Possible .............................................. 12-4
      Make Loops as Efficient as Possible ............................................................ 12-5
      Use Built-In String Functions ...................................................................... 12-5
      Put Least Expensive Conditional Tests First .............................................. 12-5
      Minimize Data Type Conversions ............................................................... 12-5
      Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic ............... 12-6
      Use BINARY_FLOAT, BINARY_DOUBLE, SIMPLE_FLOAT, and SIMPLE_DOUBLE for Floating-Point Arithmetic ................................................................. 12-6
   Avoiding Memory Overhead in PL/SQL Code .................................................. 12-7
      Declare VARCHAR2 Variables of 4000 or More Characters ...................... 12-7
      Group Related Subprograms into Packages .............................................. 12-7
      Pin Packages in the Shared Memory Pool ................................................... 12-7
      Apply Advice of Compiler Warnings ........................................................... 12-7
Collecting Data About User-Defined Identifiers .............................................. 12-7
Profiling and Tracing PL/SQL Programs ............................................................. 12-8
   Using the Profiler API: Package DBMS_PROFILER ...................................... 12-8
   Using the Trace API: Package DBMS_TRACE ............................................... 12-9
Reducing Loop Overhead for DML Statements and Queries with Bulk SQL ...... 12-9
   Running One DML Statement Multiple Times (FORALL Statement) ............ 12-10
13 PL/SQL Language Elements

Assignment Statement ................................................................. 13-3
AUTONOMOUS_TRANSACTIONPragma ......................................... 13-6
Block ......................................................................................... 13-8
CASE Statement ........................................................................ 13-17
CLOSE Statement ...................................................................... 13-20
Collection .................................................................................. 13-22
Collection Method Call ............................................................... 13-27
Comment ................................................................................... 13-31
Constant .................................................................................... 13-32
CONTINUE Statement ................................................................. 13-35
Cursor Attribute ......................................................................... 13-37
Cursor Variable Declaration ...................................................... 13-39
EXCEPTION_INITPragma ........................................................... 13-43
Exception Declaration ............................................................... 13-44
### 14 SQL Statements for Stored PL/SQL Units

<table>
<thead>
<tr>
<th>Statement</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER FUNCTION Statement</td>
<td>14-3</td>
</tr>
<tr>
<td>ALTER PACKAGE Statement</td>
<td>14-6</td>
</tr>
<tr>
<td>ALTER PROCEDURE Statement</td>
<td>14-9</td>
</tr>
<tr>
<td>ALTER TRIGGER Statement</td>
<td>14-12</td>
</tr>
<tr>
<td>ALTER TYPE Statement</td>
<td>14-15</td>
</tr>
<tr>
<td>CREATE FUNCTION Statement</td>
<td>14-29</td>
</tr>
<tr>
<td>CREATE PACKAGE Statement</td>
<td>14-39</td>
</tr>
<tr>
<td>CREATE PACKAGE BODY Statement</td>
<td>14-42</td>
</tr>
<tr>
<td>CREATE PROCEDURE Statement</td>
<td>14-46</td>
</tr>
<tr>
<td>CREATE TRIGGER Statement</td>
<td>14-51</td>
</tr>
<tr>
<td>CREATE TYPE Statement</td>
<td>14-65</td>
</tr>
<tr>
<td>CREATE TYPE BODY Statement</td>
<td>14-83</td>
</tr>
<tr>
<td>DROP FUNCTION Statement</td>
<td>14-89</td>
</tr>
<tr>
<td>DROP PACKAGE Statement</td>
<td>14-91</td>
</tr>
</tbody>
</table>
## List of Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>PL/SQL Block Structure</td>
</tr>
<tr>
<td>1–2</td>
<td>PL/SQL Variable Declarations</td>
</tr>
<tr>
<td>1–3</td>
<td>Assigning Values to Variables with the Assignment Operator</td>
</tr>
<tr>
<td>1–4</td>
<td>Using SELECT INTO to Assign Values to Variables</td>
</tr>
<tr>
<td>1–5</td>
<td>Assigning Values to Variables as Parameters of a Subprogram</td>
</tr>
<tr>
<td>1–6</td>
<td>Using %ROWTYPE with an Explicit Cursor</td>
</tr>
<tr>
<td>1–7</td>
<td>Using a PL/SQL Collection Type</td>
</tr>
<tr>
<td>1–8</td>
<td>Declaring a Record Type</td>
</tr>
<tr>
<td>1–9</td>
<td>Defining an Object Type</td>
</tr>
<tr>
<td>1–10</td>
<td>Using the IF-THEN-ELSE and CASE Statement for Conditional Control</td>
</tr>
<tr>
<td>1–11</td>
<td>Using the FOR-LOOP</td>
</tr>
<tr>
<td>1–12</td>
<td>Using WHILE-LOOP for Control</td>
</tr>
<tr>
<td>1–13</td>
<td>Using the EXIT-WHEN Statement</td>
</tr>
<tr>
<td>1–14</td>
<td>Using the GOTO Statement</td>
</tr>
<tr>
<td>1–15</td>
<td>PL/SQL Procedure</td>
</tr>
<tr>
<td>1–16</td>
<td>Creating a Standalone PL/SQL Procedure</td>
</tr>
<tr>
<td>1–17</td>
<td>Invoking a Standalone Procedure from SQL*Plus</td>
</tr>
<tr>
<td>1–18</td>
<td>Creating a Trigger</td>
</tr>
<tr>
<td>1–19</td>
<td>Creating a Package and Package Body</td>
</tr>
<tr>
<td>1–20</td>
<td>Invoking a Procedure in a Package</td>
</tr>
<tr>
<td>1–21</td>
<td>Processing Query Results in a LOOP</td>
</tr>
<tr>
<td>2–1</td>
<td>NUMBER Literals</td>
</tr>
<tr>
<td>2–2</td>
<td>Using BINARY_FLOAT and BINARY_DOUBLE</td>
</tr>
<tr>
<td>2–3</td>
<td>Using DateTime Literals</td>
</tr>
<tr>
<td>2–4</td>
<td>Single-Line Comments</td>
</tr>
<tr>
<td>2–5</td>
<td>Multiline Comment</td>
</tr>
<tr>
<td>2–6</td>
<td>Declaring Variables</td>
</tr>
<tr>
<td>2–7</td>
<td>Declaring Constants</td>
</tr>
<tr>
<td>2–8</td>
<td>Assigning Default Values to Variables with DEFAULT Keyword</td>
</tr>
<tr>
<td>2–9</td>
<td>Declaring Variables with NOT NULL Constraint</td>
</tr>
<tr>
<td>2–10</td>
<td>Using %TYPE to Declare Variables of the Types of Other Variables</td>
</tr>
<tr>
<td>2–11</td>
<td>Using %TYPE Incorrectly with NOT NULL Referenced Type</td>
</tr>
<tr>
<td>2–12</td>
<td>Using %TYPE Correctly with NOT NULL Referenced Type</td>
</tr>
<tr>
<td>2–13</td>
<td>Using %TYPE to Declare Variables of the Types of Table Columns</td>
</tr>
<tr>
<td>2–14</td>
<td>Using %ROWTYPE to Declare a Record that Represents a Table Row</td>
</tr>
<tr>
<td>2–15</td>
<td>Declaring a Record that Represents a Subset of Table Columns</td>
</tr>
<tr>
<td>2–16</td>
<td>Declaring a Record that Represents a Row from a Join</td>
</tr>
<tr>
<td>2–17</td>
<td>Assigning One Record to Another, Correctly and Incorrectly</td>
</tr>
<tr>
<td>2–18</td>
<td>Using SELECT INTO for Aggregate Assignment</td>
</tr>
<tr>
<td>2–19</td>
<td>Using an Alias for an Expression Associated with %ROWTYPE</td>
</tr>
<tr>
<td>2–20</td>
<td>Duplicate Identifiers in Same Scope</td>
</tr>
<tr>
<td>2–21</td>
<td>Case Insensitivity of Identifiers</td>
</tr>
<tr>
<td>2–22</td>
<td>Using a Block Label for Name Resolution</td>
</tr>
<tr>
<td>2–23</td>
<td>Using a Subprogram Name for Name Resolution</td>
</tr>
<tr>
<td>2–24</td>
<td>Scope and Visibility of Identifiers</td>
</tr>
<tr>
<td>2–25</td>
<td>Qualifying a Redeclared Global Identifier with a Block Label</td>
</tr>
<tr>
<td>2–26</td>
<td>Qualifying an Identifier with a Subprogram Name</td>
</tr>
<tr>
<td>2–27</td>
<td>Label and Subprogram with Same Name in Same Scope</td>
</tr>
<tr>
<td>2–28</td>
<td>Block with Multiple and Duplicate Labels</td>
</tr>
<tr>
<td>2–29</td>
<td>Variable Initialized to NULL by Default</td>
</tr>
<tr>
<td>2–30</td>
<td>Assigning BOOLEAN Values</td>
</tr>
<tr>
<td>2–31</td>
<td>Assigning Query Results to Variables</td>
</tr>
<tr>
<td>2–32</td>
<td>Concatenation Operator</td>
</tr>
<tr>
<td>2–33</td>
<td>Operator Precedence</td>
</tr>
<tr>
<td>Page</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>2-34</td>
<td>AND Operator</td>
</tr>
<tr>
<td>2-35</td>
<td>OR Operator</td>
</tr>
<tr>
<td>2-36</td>
<td>NOT Operator</td>
</tr>
<tr>
<td>2-37</td>
<td>Changing Order of Evaluation of Logical Operators</td>
</tr>
<tr>
<td>2-38</td>
<td>Short-Circuit Evaluation</td>
</tr>
<tr>
<td>2-39</td>
<td>Relational Operators</td>
</tr>
<tr>
<td>2-40</td>
<td>LIKE Operator</td>
</tr>
<tr>
<td>2-41</td>
<td>Escape Character in Pattern</td>
</tr>
<tr>
<td>2-42</td>
<td>BETWEEN Operator</td>
</tr>
<tr>
<td>2-43</td>
<td>IN Operator</td>
</tr>
<tr>
<td>2-44</td>
<td>Using the IN Operator with Sets with NULL Values</td>
</tr>
<tr>
<td>2-45</td>
<td>Using BOOLEAN Variables in Conditional Tests</td>
</tr>
<tr>
<td>2-46</td>
<td>Using the WHEN Clause with a CASE Statement</td>
</tr>
<tr>
<td>2-47</td>
<td>Using a Search Condition with a CASE Statement</td>
</tr>
<tr>
<td>2-48</td>
<td>NULL Value in Unequal Comparison</td>
</tr>
<tr>
<td>2-49</td>
<td>NULL Value in Equal Comparison</td>
</tr>
<tr>
<td>2-50</td>
<td>NULL Value as Argument to DECODE Function</td>
</tr>
<tr>
<td>2-51</td>
<td>NULL Value as Argument to NVL Function</td>
</tr>
<tr>
<td>2-52</td>
<td>NULL Value as Second Argument to REPLACE Function</td>
</tr>
<tr>
<td>2-53</td>
<td>NULL Value as Third Argument to REPLACE Function</td>
</tr>
<tr>
<td>2-54</td>
<td>Using Static Constants</td>
</tr>
<tr>
<td>2-55</td>
<td>Using DBMS_DB_VERSION Constants</td>
</tr>
<tr>
<td>2-56</td>
<td>Using Conditional Compilation with Database Versions</td>
</tr>
<tr>
<td>2-57</td>
<td>Using PRINT_POST_PROCESSED_SOURCE to Display Source Code</td>
</tr>
<tr>
<td>3-1</td>
<td>Comparing Two CHAR Values</td>
</tr>
<tr>
<td>3-2</td>
<td>Comparing Two VARCHAR2 Values</td>
</tr>
<tr>
<td>3-3</td>
<td>Comparing CHAR Value and VARCHAR2 Value</td>
</tr>
<tr>
<td>3-4</td>
<td>Assigning a Literal Value to a TIMESTAMP Variable</td>
</tr>
<tr>
<td>3-5</td>
<td>Using the SCN_TO_TIMESTAMP and TIMESTAMP_TO_SCN Functions</td>
</tr>
<tr>
<td>3-6</td>
<td>Assigning a Literal to a TIMESTAMP WITH TIME ZONE Variable</td>
</tr>
<tr>
<td>3-7</td>
<td>Correct Assignment to TIMESTAMP WITH LOCAL TIME ZONE</td>
</tr>
<tr>
<td>3-8</td>
<td>Incorrect Assignment to TIMESTAMP WITH LOCAL TIME ZONE</td>
</tr>
<tr>
<td>3-9</td>
<td>Assigning Literals to an INTERVAL YEAR TO MONTH Variable</td>
</tr>
<tr>
<td>3-10</td>
<td>Assigning Literals to an INTERVAL DAY TO SECOND Variable</td>
</tr>
<tr>
<td>3-11</td>
<td>Using Ranges with Subtypes</td>
</tr>
<tr>
<td>3-12</td>
<td>Type Compatibility with the NUMBER Data Type</td>
</tr>
<tr>
<td>3-13</td>
<td>Assigning Default Value to Subtype Variable</td>
</tr>
<tr>
<td>3-14</td>
<td>Subtype Constraints Inherited by Subprograms</td>
</tr>
<tr>
<td>3-15</td>
<td>Column Constraints Inherited by Subtypes</td>
</tr>
<tr>
<td>3-16</td>
<td>Implicit Conversion</td>
</tr>
<tr>
<td>4-1</td>
<td>Simple IF-THEN Statement</td>
</tr>
<tr>
<td>4-2</td>
<td>Using a Simple IF-THEN-ELSE Statement</td>
</tr>
<tr>
<td>4-3</td>
<td>Nested IF-THEN-ELSE Statements</td>
</tr>
<tr>
<td>4-4</td>
<td>Using the IF-THEN-ELSIF Statement</td>
</tr>
<tr>
<td>4-5</td>
<td>Extended IF-THEN Statement</td>
</tr>
<tr>
<td>4-6</td>
<td>Simple CASE Statement</td>
</tr>
<tr>
<td>4-7</td>
<td>Searched CASE Statement</td>
</tr>
<tr>
<td>4-8</td>
<td>Using EXCEPTION Instead of ELSE Clause in CASE Statement</td>
</tr>
<tr>
<td>4-9</td>
<td>EXIT Statement</td>
</tr>
<tr>
<td>4-10</td>
<td>Using an EXIT-WHEN Statement</td>
</tr>
<tr>
<td>4-11</td>
<td>CONTINUE Statement</td>
</tr>
<tr>
<td>4-12</td>
<td>CONTINUE-WHEN Statement</td>
</tr>
<tr>
<td>4-13</td>
<td>Labeled Loops</td>
</tr>
<tr>
<td>4-14</td>
<td>Simple FOR-LOOP Statement</td>
</tr>
<tr>
<td>4-15</td>
<td>Reverse FOR-LOOP Statement</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6-39</td>
<td>How Invalid Subscripts are Handled with DELETE(n)</td>
</tr>
<tr>
<td>6-40</td>
<td>Incompatibility Between Package and Local Collection Types</td>
</tr>
<tr>
<td>6-41</td>
<td>Declaring and Initializing a Simple Record Type</td>
</tr>
<tr>
<td>6-42</td>
<td>Declaring and Initializing Record Types</td>
</tr>
<tr>
<td>6-43</td>
<td>Using %ROWTYPE to Declare a Record</td>
</tr>
<tr>
<td>6-44</td>
<td>Returning a Record from a Function</td>
</tr>
<tr>
<td>6-45</td>
<td>Using a Record as Parameter to a Procedure</td>
</tr>
<tr>
<td>6-46</td>
<td>Declaring a Nested Record</td>
</tr>
<tr>
<td>6-47</td>
<td>Assigning Default Values to a Record</td>
</tr>
<tr>
<td>6-48</td>
<td>Assigning All the Fields of a Record in One Statement</td>
</tr>
<tr>
<td>6-49</td>
<td>Using SELECT INTO to Assign Values in a Record</td>
</tr>
<tr>
<td>6-50</td>
<td>Inserting a PL/SQL Record Using %ROWTYPE</td>
</tr>
<tr>
<td>6-51</td>
<td>Updating a Row Using a Record</td>
</tr>
<tr>
<td>6-52</td>
<td>Using the RETURNING INTO Clause with a Record</td>
</tr>
<tr>
<td>6-53</td>
<td>Using BULK COLLECT with a SELECT INTO Statement</td>
</tr>
<tr>
<td>6-1</td>
<td>Data Manipulation with PL/SQL</td>
</tr>
<tr>
<td>6-2</td>
<td>Checking SQL%ROWCOUNT After an UPDATE</td>
</tr>
<tr>
<td>6-3</td>
<td>Substituting PL/SQL Variables</td>
</tr>
<tr>
<td>6-4</td>
<td>Invoking the SQL COUNT Function in PL/SQL</td>
</tr>
<tr>
<td>6-5</td>
<td>Using CURRVAL and NEXTVAL</td>
</tr>
<tr>
<td>6-6</td>
<td>Using ROWNUM</td>
</tr>
<tr>
<td>6-7</td>
<td>Using SQL%FOUND</td>
</tr>
<tr>
<td>6-8</td>
<td>Using SQL%ROWCOUNT</td>
</tr>
<tr>
<td>6-9</td>
<td>Declaring a Cursor</td>
</tr>
<tr>
<td>6-10</td>
<td>Fetching with a Cursor</td>
</tr>
<tr>
<td>6-11</td>
<td>Referencing PL/SQL Variables Within Its Scope</td>
</tr>
<tr>
<td>6-12</td>
<td>Fetching the Same Cursor Into Different Variables</td>
</tr>
<tr>
<td>6-13</td>
<td>Fetching Bulk Data with a Cursor</td>
</tr>
<tr>
<td>6-14</td>
<td>Using %FOUND</td>
</tr>
<tr>
<td>6-15</td>
<td>Using %ISOPEN</td>
</tr>
<tr>
<td>6-16</td>
<td>Using %NOTFOUND</td>
</tr>
<tr>
<td>6-17</td>
<td>Using %ROWCOUNT</td>
</tr>
<tr>
<td>6-18</td>
<td>Using an Alias For Expressions in a Query</td>
</tr>
<tr>
<td>6-19</td>
<td>Using a Subquery in a Cursor</td>
</tr>
<tr>
<td>6-20</td>
<td>Using a Subquery in a FROM Clause</td>
</tr>
<tr>
<td>6-21</td>
<td>Using a Correlated Subquery</td>
</tr>
<tr>
<td>6-22</td>
<td>Passing Parameters to a Cursor FOR Loop</td>
</tr>
<tr>
<td>6-23</td>
<td>Passing Parameters to Explicit Cursors</td>
</tr>
<tr>
<td>6-24</td>
<td>Cursor Variable Returning a %ROWTYPE Variable</td>
</tr>
<tr>
<td>6-25</td>
<td>Using the %ROWTYPE Attribute to Provide the Data Type</td>
</tr>
<tr>
<td>6-26</td>
<td>Cursor Variable Returning a Record Type</td>
</tr>
<tr>
<td>6-27</td>
<td>Passing a REF CURSOR as a Parameter</td>
</tr>
<tr>
<td>6-28</td>
<td>Checking If a Cursor Variable is Open</td>
</tr>
<tr>
<td>6-29</td>
<td>Stored Procedure to Open a Ref Cursor</td>
</tr>
<tr>
<td>6-30</td>
<td>Stored Procedure to Open Ref Cursors with Different Queries</td>
</tr>
<tr>
<td>6-31</td>
<td>Cursor Variable with Different Return Types</td>
</tr>
<tr>
<td>6-32</td>
<td>Fetching from a Cursor Variable into a Record</td>
</tr>
<tr>
<td>6-33</td>
<td>Fetching from a Cursor Variable into Collections</td>
</tr>
<tr>
<td>6-34</td>
<td>Declaration of Cursor Variables in a Package</td>
</tr>
<tr>
<td>6-35</td>
<td>Using a Cursor Expression</td>
</tr>
<tr>
<td>6-36</td>
<td>Using COMMIT with the WRITE Clause</td>
</tr>
<tr>
<td>6-37</td>
<td>Using ROLLBACK</td>
</tr>
<tr>
<td>6-38</td>
<td>Using SAVEPOINT with ROLLBACK</td>
</tr>
<tr>
<td>6-39</td>
<td>reusing a SAVEPOINT with ROLLBACK</td>
</tr>
</tbody>
</table>
Using the Dot Notation to Qualify Names
List of Figures

1–1  PL/SQL Boosts Performance ................................................................. 1-2
1–2  PL/SQL Engine ................................................................................. 1-24
4–1  Control Structures .......................................................................... 4-1
5–1  Array and Nested Table ................................................................. 5-5
5–2  Varray of Size 10 ........................................................................ 5-5
6–1  Transaction Control Flow ............................................................... 6-41
8–1  How the PL/SQL Compiler Resolves Calls ........................................... 8-17
10–1 Package Scope ................................................................................ 10-4
11–1 Propagation Rules: Example 1 ......................................................... 11-11
11–2 Propagation Rules: Example 2 ......................................................... 11-11
11–3 Propagation Rules: Example 3 ......................................................... 11-12
**List of Tables**

1–1    PL/SQL Compilation Parameters ................................................................. 1-26
2–1    PL/SQL Delimiters .......................................................................................... 2-3
2–2    Operator Precedence ....................................................................................... 2-29
2–3    Logical Truth Table ....................................................................................... 2-30
2–4    Relational Operators ...................................................................................... 2-35
3–1    Categories of Predefined PL/SQL Data Types ................................................ 3-1
3–2    Categories of Predefined PL/SQL Scalar Data Types ....................................... 3-2
3–3    Predefined PL/SQL Numeric Data Types ......................................................... 3-2
3–4    Predefined Subtypes of PLS_INTEGER Data Type ............................................. 3-3
3–5    Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants ............. 3-5
3–6    Predefined Subtypes of NUMBER Data Type .................................................. 3-7
3–7    Predefined PL/SQL Character Data Types ...................................................... 3-7
3–8    Comparison of AL16UTF16 and UTF8 Encodings .......................................... 3-13
3–9    Predefined PL/SQL Large Object (LOB) Data Types ....................................... 3-22
3–10   Possible Implicit PL/SQL Data Type Conversions .......................................... 3-31
5–1    Characteristics of PL/SQL Collection Types .................................................. 5-2
6–1    Cursor Attribute Values ................................................................................... 6-15
8–1    Parameter Modes ............................................................................................ 8-9
8–2    PL/SQL Subprogram Parameter Notations ....................................................... 8-11
8–3    Comparison of Finer and Coarser Caching Granularity ................................... 8-35
9–1    Timing-Point Sections of a Compound Trigger Defined .................................... 9-15
9–2    Comparison of Built-in Auditing and Trigger-Based Auditing ........................... 9-31
9–3    System-Defined Event Attributes .................................................................... 9-47
9–4    Database Events ............................................................................................. 9-50
9–5    Client Events .................................................................................................. 9-51
11–1   Predefined PL/SQL Exceptions ...................................................................... 11-4
11–2   PL/SQL Warning Categories ......................................................................... 11-19
C–1    PL/SQL Compiler Limits ............................................................................... C-1
D–1    PL/SQL Reserved Words ................................................................................ D-1
D–2    PL/SQL Keywords .......................................................................................... D-2
Preface

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

Preface topics:
- Audience
- Documentation Accessibility
- Related Documents
- Conventions
- Syntax Descriptions

Audience

Oracle Database PL/SQL Language Reference is intended for anyone who is developing PL/SQL-based applications for an 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, 7 days a week. For TTY support, call 800.446.2398. Outside the United States, call +1.407.458.2479.

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:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><em>monospace</em></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
<tr>
<td>{A</td>
<td>B</td>
</tr>
</tbody>
</table>

*-_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 Language Reference for information about how to interpret these descriptions.
This topic briefly describes the new PL/SQL features that this book documents and provides links to more 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`, `SIMPLE_FLOAT`, and `SIMPLE_DOUBLE` Data Types
- `CONTINUE` Statement
- Sequences in PL/SQL Expressions
- Dynamic SQL Enhancements
- Named and Mixed Notation in PL/SQL Subprogram Invocations
- PL/SQL Function Result Cache
- Compound Triggers
- More Control Over Triggers
- Database Resident Connection Pool
- Automatic Subprogram Inlining
- PL/Scope
- PL/SQL Hierarchical Profiler
- PL/SQL Native Compiler Generates Native Code Directly

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

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

**See Also:**

- *Oracle Database Advanced Application Developer’s Guide* for information about the implementation of regular expressions
- *Oracle Database SQL Language Reference* for detailed descriptions of the `REGEXP_INSTR`, `REGEXP_SUBSTR`, and `REGEXP_COUNT` functions
SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE Data Types

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

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

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

For more information, see:
- SIMPLE_INTEGER Subtype of PLS_INTEGER on page 3-3
- BINARY_FLOAT and BINARY_DOUBLE Data Types on page 3-5
- Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic on page 12-6
- Use BINARY_FLOAT, BINARY_DOUBLE, SIMPLE_FLOAT, and SIMPLE_DOUBLE for Floating-Point Arithmetic on page 12-6

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 4-8
- CONTINUE Statement on page 13-35

Sequences in PL/SQL Expressions

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

For more information, see CURRVAL and NEXTVAL on page 6-4.

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 13-47 and OPEN-FOR Statement on page 13-96.

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

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

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

For an example, see the SELECT statements following Example 8–8 on page 8-11.

**PL/SQL Function Result Cache**

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

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

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

If you convert your application to PL/SQL function result caching, your application will use more SGA, but significantly less total system memory.

For more information, see:
- Using the PL/SQL Function Result Cache on page 8-27
- Table , 'Function Declaration and Definition' on page 13-75

**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 9-13.

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

- Ordering of Triggers on page 9-8
- Enabling Triggers on page 9-29
- Disabling Triggers on page 9-29

See Also: CREATE TRIGGER Statement on page 14-51

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
PL/SQL unit) with a copy of the called subprogram, which almost always improves
program performance.
You can use PRAGMA INLINE to specify that individual subprogram calls are, or are not, to be inlined. You can also turn on automatic inlining—that is, ask the compiler to
search for inlining opportunities—by setting the compilation parameter PLSQL_OPTIMIZE_LEVEL to 3 (the default is 2).
In the rare cases when automatic inlining does not improve program performance, you
can use the PL/SQL hierarchical profiler to identify subprograms for which you want
to turn off inlining.
For more information, see:

- How PL/SQL Optimizes Your Programs on page 12-1
- INLINE Pragma on page 13-82

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

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

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

For more information, see Collecting Data About User-Defined Identifiers on page 12-7.

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 12-8.

See Also: Oracle Database Advanced Application Developer’s Guide

PL/SQL Native Compiler Generates Native Code Directly

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

For more information, see Compiling PL/SQL Units for Native Execution on page 12-30.
Overview of PL/SQL

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

Topics:

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

Advantages of PL/SQL

PL/SQL has these advantages:

■ Tight Integration with SQL
■ High Performance
■ High Productivity
■ Full Portability
■ Tight Security
■ Access to Predefined Packages
■ Support for Object-Oriented Programming
■ Support for Developing Web Applications and Server 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 is tightly integrated with SQL. With PL/SQL, you can use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, operators, and pseudocolumns.

PL/SQL fully supports SQL data types. You need not convert between PL/SQL and SQL data types. For example, if your PL/SQL program retrieves a value from a database column of the SQL type VARCHAR2, it can store that value in a PL/SQL variable of the type VARCHAR2. Special PL/SQL language features let you work with table columns and rows without specifying the data types, 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."

**High Performance**

With PL/SQL, an entire block of statements can be sent to the database 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**

**High 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,
Advantages of PL/SQL

Overview of PL/SQL

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 data types.

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 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 about 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 triggers, DBMS_FILE for reading and writing operating system text files, UTL_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 more information about these packages, see Overview of Product-Specific PL/SQL Packages on page 10-10.

For complete information about 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 about object types, see Oracle Database Object-Relational Developer’s Guide.
Support for Developing Web Applications and Server Pages

You can use PL/SQL to develop Web applications and Server Pages (PSPs). For more information, see Using PL/SQL to Create Web Applications on page 2-56 and Using PL/SQL to Create Server Pages on page 2-57.

Main Features of PL/SQL

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

When a problem can be solved using SQL, you can issue SQL statements from your PL/SQL programs, without learning new APIs.

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

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

Topics:

- PL/SQL Blocks
- PL/SQL Error Handling
- PL/SQL Input and Output
- PL/SQL Variables and Constants
- PL/SQL Data Abstraction
- PL/SQL Control Structures
- PL/SQL Subprograms
- PL/SQL Packages (APIs Written in PL/SQL)
- Conditional Compilation
- Embedded SQL Statements

PL/SQL Blocks

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

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

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

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

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

Example 1–1 PL/SQL Block Structure

```sql
DECLARE
  -- Declarative part (optional)
  -- Declarations of local types, variables, & subprograms
END;
```
BEGIN -- Executable part (required)
   -- Statements (which can use items declared in declarative part)

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

A PL/SQL block can be submitted to an interactive tool (such as SQL*Plus or Enterprise Manager) or embedded in an Oracle Precompiler or OCI program. The interactive tool or program executes the block only once. The block is not stored in the database.

A named PL/SQL block—a subprogram—can be invoked repeatedly (see PL/SQL Subprograms on page 1-18).

---

Note: A block that is not stored in the database is called an anonymous block, even if it has a label.

---

PL/SQL Error Handling

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

PL/SQL exception handling differs from the manual checking that you do in C programming, where you insert a check to 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 define 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 raise an exception with either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR. See the exception comm_missing in Example 1–16 on page 1-18. 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 5-29.

For information about PL/SQL errors, see Overview of PL/SQL Run-Time Error Handling on page 11-1. For information about PL/SQL warnings, see Overview of PL/SQL Compile-Time Warnings on page 11-19.
PL/SQL Input and Output

Most PL/SQL input and output (I/O) is through SQL statements that store data in database tables or query those tables. All other PL/SQL I/O is done through APIs, such as the PL/SQL package DBMS_OUTPUT.

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

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

<table>
<thead>
<tr>
<th>Package(s)</th>
<th>PL/SQL uses package ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTF and HTP</td>
<td>to display output on a web page</td>
</tr>
<tr>
<td>DBMS_PIPE</td>
<td>to pass information between PL/SQL and operating-system commands</td>
</tr>
<tr>
<td>UTL_FILE</td>
<td>to read and write operating system files</td>
</tr>
<tr>
<td>UTL_HTTP</td>
<td>to communicate with web servers</td>
</tr>
<tr>
<td>UTL_SMTP</td>
<td>to communicate with mail servers</td>
</tr>
</tbody>
</table>

Although some of the preceding APIs can accept input as well as display output, they have cannot accept data directly from the keyboard. For that, use the SQL*Plus commands PROMPT and ACCEPT.

See Also:
- SQL*Plus User’s Guide and Reference for information about the SQL*Plus command PROMPT
- SQL*Plus User’s Guide and Reference for information about the SQL*Plus command ACCEPT
- Oracle Database PL/SQL Packages and Types Reference for detailed information about all PL/SQL packages

PL/SQL Variables and Constants

PL/SQL lets you declare variables and constants, and then use them in SQL and procedural statements anywhere an expression can be used. You must declare a variable or constant before referencing it in any other statements. For more information, see Declarations on page 2-10.

Topics:
- Declaring PL/SQL Variables
- Assigning Values to Variables
- Declaring PL/SQL Constants
- Bind Variables

Declaring PL/SQL Variables

A PL/SQL variable can have any SQL data type (such as CHAR, DATE, or NUMBER) or a PL/SQL-only data type (such as BOOLEAN or PLS_INTEGER).

Example 1–2 declares several PL/SQL variables. One has a PL/SQL-only data type; the others have SQL data types.
Example 1–2  PL/SQL Variable Declarations

```
SQL> DECLARE
2  part_number       NUMBER(6);     -- SQL data type
3  part_name         VARCHAR2(20);  -- SQL data type
4  in_stock          BOOLEAN;       -- PL/SQL-only data type
5  part_price        NUMBER(6,2);   -- SQL data type
6  part_description  VARCHAR2(50);  -- SQL data type
7  BEGIN
8  END;
9  /
```

PL/SQL procedure successfully completed.

SQL>

For more information about PL/SQL data types, see Chapter 3, “PL/SQL Data Types.”

PL/SQL also lets you declare composite data types, such as nested tables, variable-size arrays, and records. For more informations, see Chapter 5, “Using PL/SQL Collections and Records.”

Assigning Values to Variables
You can assign a value to a variable in the following ways:
- With the assignment operator (:=), as in Example 1–3.
- By selecting (or fetching) database values into it, as in Example 1–4.
- By passing it as an OUT or IN OUT parameter to a subprogram, and then assigning the value inside the subprogram, as in Example 1–5

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

```
SQL> DECLARE
2  wages          NUMBER;
3  hours_worked   NUMBER := 40;
4  hourly_salary  NUMBER := 22.50;
5  bonus          NUMBER := 150;
6  country        VARCHAR2(128);
7  counter        NUMBER := 0;
8  done           BOOLEAN;
9  valid_id       BOOLEAN;
10  emp_rec1       employees%ROWTYPE;
11  emp_rec2       employees%ROWTYPE;
12  TYPE commissions IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
13  comm_tab       commissions;
14
15  BEGIN
16  wages := (hours_worked * hourly_salary) + bonus;
17  country := 'France';
18  country := UPPER('Canada');
19  done := (counter > 100);
20  valid_id := TRUE;
21  emp_rec1.first_name := 'Antonio';
22  emp_rec1.last_name := 'Ortiz';
23  emp_rec1 := emp_rec2;
24  comm_tab(5) := 20000 * 0.15;
25  END;
26  /
```
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

Example 1–5 passes the new_sal variable to a subprogram, and the subprogram updates the variable.

Example 1–5  Assigning Values to Variables as Parameters of a Subprogram
DECLARE
    credit_limit CONSTANT NUMBER := 5000.00;

END;

The average salary for all employees: 6461.68
The average salary for ST_CLERK employees: 2785

PL/SQL procedure successfully completed.

Declarating PL/SQL Constants
Declaring a PL/SQL constant is like declaring a PL/SQL variable except that you must add the keyword CONSTANT and immediately assign a value to the constant. For example:

credit_limit CONSTANT NUMBER := 5000.00;

No further assignments to the constant are allowed.

Bind Variables
Bind variables improve performance by allowing the database 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. The database 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.

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
- %TYPE Attribute
- %ROWTYPE Attribute
- 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 1–6 on page 1-11 declares an explicit cursor.

For information about cursors, see Managing Cursors in PL/SQL on page 6-7.

%TYPE Attribute

The %TYPE attribute provides the data type 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 data type 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 data type of last_name. Second, if you change the database definition of last_name, perhaps to make it a longer character string, the data type of v_last_name changes accordingly at run time.

For more information about %TYPE, see Using the %TYPE Attribute on page 2-12 and %TYPE Attribute on page 13-128.

%ROWTYPE Attribute

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 1-10.

Columns in a row and corresponding fields in a record have the same names and data types. In the following example, you declare a record named dept_rec, whose fields have the same names and data types as the columns in the departments table:

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

You use dot notation to reference fields, as follows:

```
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. The FETCH statement in Example 1–6 assigns the value in the last_name column of the employees table to the last_name field of employee_rec, the value in the salary column is to the salary field, and so on.

Example 1–6  Using %ROWTYPE with an Explicit Cursor

```
SQL> DECLARE
2    CURSOR c1 IS
3      SELECT last_name, salary, hire_date, job_id
4        FROM employees
5          WHERE employee_id = 120;
6```
Main Features of PL/SQL

Overview of PL/SQL

7    employee_rec c1%ROWTYPE;
8
9 BEGIN
10   OPEN c1;
11   FETCH c1 INTO employee_rec;
12   DBMS_OUTPUT.PUT_LINE('Employee name: ' || employee_rec.last_name);
13 END;
14 /
Employee name: Weiss

PL/SQL procedure successfully completed.

SQL>

For more information about %ROWTYPE, see Using the %ROWTYPE Attribute on page 2-15 and %ROWTYPE Attribute on page 13-114.

Collections

PL/SQL collection types let you declare high-level data types 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 1–7.

Example 1–7 Using a PL/SQL Collection Type

7   TYPE staff_list IS TABLE OF employees.employee_id%TYPE;
8   staff  staff_list;
9   lname  employees.last_name%TYPE;
10  fname  employees.first_name%TYPE;
11 BEGIN
12   staff := staff_list(100, 114, 115, 120, 122);
13   FOR i IN staff.FIRST..staff.LAST LOOP
14      SELECT last_name, first_name INTO lname, fname
15          FROM employees
16          WHERE employees.employee_id = staff(i);
17      DBMS_OUTPUT.PUT_LINE (TO_CHAR(staff(i))
18          || ' : ' || lname
19          || ', ' || fname
20          );
21 END LOOP;
22 END;
23 /
100: King, Steven
114: Raphaely, Den
115: Khoo, Alexander
120: Weiss, Matthew
122: Kaufling, Payam
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 about collections, see Chapter 5, "Using PL/SQL Collections and Records."

**Records**

Records are composite data structures whose fields can have different data types. 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, as in Example 1–8. See Defining and Declaring Records on page 5-31.

**Example 1–8 Declaring a Record Type**

```sql
SQL> DECLARE
2    TYPE timerec IS RECORD (
3      hours   SMALLINT,
4      minutes SMALLINT
5    );
6
7    TYPE meeting_type IS RECORD (
8      date_held  DATE,
9      duration   timerec, -- nested record
10     location   VARCHAR2(20),
11     purpose    VARCHAR2(50)
12    );
13
14  BEGIN
15    NULL;
16  END;
17 /

PL/SQL procedure successfully completed.
```

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 about 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 1–9 shows an object type definition for a bank account.
**Example 1–9  Defining an Object Type**

```sql
SQL> CREATE TYPE bank_account AS OBJECT {
  2    acct_number NUMBER(5),
  3    balance NUMBER,
  4    status VARCHAR2(10),
  5
  6    MEMBER PROCEDURE open
  7      (SELF IN OUT NOCOPY bank_account,
  8        amount IN NUMBER),
  9
  10   MEMBER PROCEDURE close
  11     (SELF IN OUT NOCOPY bank_account,
  12       num IN NUMBER,
  13       amount OUT NUMBER),
  14
  15   MEMBER PROCEDURE deposit
  16     (SELF IN OUT NOCOPY bank_account,
  17       num IN NUMBER,
  18       amount IN NUMBER),
  19
  20   MEMBER PROCEDURE withdraw
  21     (SELF IN OUT NOCOPY bank_account,
  22       num IN NUMBER,
  23       amount IN NUMBER),
  24
  25   MEMBER FUNCTION curr_bal (num IN NUMBER) RETURN NUMBER
  26  };
  27 /

Type created.

SQL>
```

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

---

**PL/SQL Control Structures**

Control structures are the most important PL/SQL extension to SQL. Not only does PL/SQL let you manipulate database data, it lets you process the data using flow-of-control statements.

Topics:
- **Conditional Control**
- **Iterative Control**
- **Sequential Control**

For more information, see Chapter 4, "Using PL/SQL Control Structures."

**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 1–10** 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 for each case, as in Example 1–10.

**Example 1–10 Using the IF-THEN-ELSE and CASE Statement for Conditional Control**

```sql
SQL> DECLARE
2     jobid  employees.job_id%TYPE;
3     empid  employees.employee_id%TYPE := 115;
4     sal    employees.salary%TYPE;
5     sal_raise NUMBER(3,2);
6  BEGIN
7    SELECT job_id, salary INTO jobid, sal
8      FROM employees
9        WHERE employee_id = empid;
10
11    CASE
12      WHEN jobid = 'PU_CLERK' THEN
13        IF sal < 3000 THEN
14          sal_raise := .12;
15        ELSE
16          sal_raise := .09;
17        END IF;
18
19      WHEN jobid = 'SH_CLERK' THEN
20        IF sal < 4000 THEN
21          sal_raise := .11;
22        ELSE
23          sal_raise := .08;
24        END IF;
25
26      WHEN jobid = 'ST_CLERK' THEN
27        IF sal < 3500 THEN
28          sal_raise := .10;
29        ELSE
30          sal_raise := .07;
31        END IF;
32
33      ELSE
34        BEGIN
35          DBMS_OUTPUT.PUT_LINE('No raise for this job: ' || jobid);
36        END;
37     END CASE;
38
39     UPDATE employees
40       SET salary = salary + salary * sal_raise
41         WHERE employee_id = empid;
42  END;
43 /

PL/SQL procedure successfully completed.

SQL>
```

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 1–11 the loop inserts 100 numbers, square roots, squares, and the sum of squares into a database table.

**Example 1–11  Using the FOR-LOOP**

```
SQL> CREATE TABLE sqr_root_sum (
    2    num NUMBER,
    3    sq_root NUMBER(6,2),
    4    sqr NUMBER,
    5    sum_sqs NUMBER
    6  );
Table created.

SQL> DECLARE
    2     s  PLS_INTEGER;
    3  BEGIN
    4    FOR i in 1..100 LOOP
    5      s := (i * (i + 1) * (2*i +1)) / 6;  -- sum of squares
    6    END LOOP;
    7    INSERT INTO sqr_root_sum
    8        VALUES (i, SQRT(i), i*i, s );
    9  END;
10  /
PL/SQL procedure successfully completed.
```

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 1–12, you find the first employee who has a salary over $15000 and is higher in the chain of command than employee 120.

**Example 1–12  Using WHILE-LOOP for Control**

```
SQL> CREATE TABLE temp (
    2    tempid   NUMBER(6),
    3    tempsal  NUMBER(8,2),
    4    tempname VARCHAR2(25)
    5  );
Table created.
```
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 1–13, 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 1–13 Using the EXIT-WHEN Statement**

```sql
SQL> CREATE TABLE temp ( 2   tempid    NUMBER(6), 3   tempsal   NUMBER(8,2), 4   tempname  VARCHAR2(25) 5 )
Table created.

SQL>
```
10  DBMS_OUTPUT.PUT_LINE ('Counter: ' || TO_CHAR(counter) || ' Total: ' || TO_CHAR(total));
11  END;
12  /

Counter: 42 Total: 25585

PL/SQL procedure successfully completed.

SQL>

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 in Example 1–14.

Example 1–14 Using the GOTO Statement

SQL> DECLARE
2    total  NUMBER(9) := 0;
3    counter NUMBER(6) := 0;
4  BEGIN
5      <<calc_total>>
6      counter := counter + 1;
7      total := total + counter * counter;
8      IF total > 25000 THEN
9        GOTO print_total;
10      ELSE
11        GOTO calc_total;
12    END IF;
13    END <<calc_total>>
14      <<print_total>>
15      DBMS_OUTPUT.PUT_LINE
16        ('Counter: ' || TO_CHAR(counter) || ' Total: ' || TO_CHAR(total));
17  END;
18  /

Counter: 42 Total: 25585

PL/SQL procedure successfully completed.

SQL>

PL/SQL Subprograms

A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters, like double in Example 1–15. PL/SQL has two types of subprograms, procedures and functions. A function returns a result.

Example 1–15 PL/SQL Procedure

SQL> DECLARE
2    in_string  VARCHAR2(100) := 'Test string';
3    out_string VARCHAR2(200);
4
PROCEDURE double (original IN VARCHAR2, new_string OUT VARCHAR2) AS
BEGIN
new_string := original || original;
END;
BEGIN
DBMS_OUTPUT.PUT_LINE ('in_string: ' || in_string);
double (in_string, out_string);
DBMS_OUTPUT.PUT_LINE ('out_string: ' || out_string);
END;
/
in_string: Test string
out_string: Test stringTest string

PL/SQL procedure successfully completed.

Topics:
- Standalone PL/SQL Subprograms
- Triggers

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

Standalone PL/SQL Subprograms
You create standalone subprograms at schema level with the SQL statements CREATE PROCEDURE and CREATE FUNCTION. They are compiled and stored in the database, where they can be used by any number of applications connected to the database. When invoked, they are loaded and processed immediately. Subprograms use shared memory, so that only one copy of a subprogram is loaded into memory for execution by multiple users.

Example 1–16 creates a standalone procedure that accepts an employee ID and a bonus amount, uses the ID to select the employee's commission percentage from a database table and to convert the commission percentage to a decimal amount, and then checks the commission amount. If the commission is null, the procedure raises an exception; otherwise, it updates the employee's salary.

Example 1–16 Creating a Standalone PL/SQL Procedure

CREATE OR REPLACE PROCEDURE award_bonus (emp_id NUMBER, bonus NUMBER) AS
commission REAL;
comm_missing EXCEPTION;
BEGIN
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
END IF;
A PL/SQL subprogram can be invoked from an interactive tool (such as SQL*Plus or Enterprise Manager), from an Oracle Precompiler or OCI program, from another PL/SQL subprogram, or from a trigger.

For information about the CREATE PROCEDURE statement, see CREATE PROCEDURE Statement on page 14-46.

For more information about the SQL CREATE FUNCTION, see CREATE FUNCTION Statement on page 14-29.

**Example 1–17** invokes the stored subprogram in **Example 1–16** with the CALL statement and then from inside a block.

**Example 1–17  Invoking a Standalone Procedure from SQL*Plus**

```sql
SQL> -- Invoke standalone procedure with CALL statement
SQL> CALL award_bonus(179, 1000);
Call completed.

SQL> -- Invoke standalone procedure from within block
SQL> BEGIN
  2  award_bonus(179, 10000);
  3 END;
  4 /
PL/SQL procedure successfully completed.
```

Using the BEGIN-END block is recommended in several situations. For example, using the CALL statement can suppress an ORA-n error that was not handled in the PL/SQL subprogram.

For additional examples of invoking PL/SQL subprograms, see Example 8–8 on page 8-11. For information about the CALL statement, see Oracle Database SQL Language Reference

**Triggers**

A trigger is a stored subprogram associated with a 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 before or after the event.

The trigger in Example 1–18 is invoked whenever salaries in the `employees` table are updated. For each update, the trigger writes a record to the `emp_audit` table. (Example 1–10 on page 1-14 would invoke this trigger.)

**Example 1–18  Creating a Trigger**

```sql
SQL> CREATE TABLE emp_audit (  
2    emp_audit_id  NUMBER(6),  
3    up_date       DATE,  
4    new_sal       NUMBER(8,2),  
5    old_sal       NUMBER(8,2)  
6  );
Table created.

SQL> CREATE OR REPLACE TRIGGER audit_sal  
2   AFTER UPDATE OF salary  
3   ON employees  
4   FOR EACH ROW  
5   BEGIN  
6     INSERT INTO emp_audit  
7     VALUES(:old.employee_id, SYSDATE, :new.salary, :old.salary);  
8   END;
9 /  
Trigger created.

SQL>
```

For more information about triggers, see Chapter 9, "Using Triggers."

**PL/SQL Packages (APIs Written in PL/SQL)**

A PL/SQL package bundles 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.

PL/SQL lets you access many predefined packages (see Access to Predefined Packages on page 1-3) and to create your own packages.

A package usually has two parts: a specification and a body.

The specification defines the application programming interface (API); it declares the types, constants, variables, exceptions, cursors, and subprograms. To create a package specification, use the `CREATE PACKAGE` Statement on page 14-39.

The body contains the SQL queries for cursors and the code for subprograms. To create a package body, use the `CREATE PACKAGE BODY` Statement on page 14-42.

In Example 1–19, the `emp_actions` package contains two procedures that update the `employees` table and one function that provides information.

**Example 1–19 Creating a Package and Package Body**

```sql
SQL> -- Package specification:
SQL> 
```
SQL> CREATE OR REPLACE PACKAGE emp_actions AS
3  PROCEDURE hire_employee ( employee_id NUMBER,
4      last_name VARCHAR2,
5      first_name VARCHAR2,
6      email VARCHAR2,
7      phone_number VARCHAR2,
8      hire_date DATE,
9      job_id VARCHAR2,
10     salary NUMBER,
11    commission_pct NUMBER,
12    manager_id NUMBER,
13    department_id NUMBER
14  );
15
16  PROCEDURE fire_employee (emp_id NUMBER);
17
18  FUNCTION num_above_salary (emp_id NUMBER) RETURN NUMBER;
19  END emp_actions;
20  /

Package created.

SQL> -- Package body:
SQL> CREATE OR REPLACE PACKAGE BODY emp_actions AS
2  -- Code for procedure hire_employee:
4  PROCEDURE hire_employee ( employee_id NUMBER,
5      last_name VARCHAR2,
6      first_name VARCHAR2,
7      email VARCHAR2,
8      phone_number VARCHAR2,
9      hire_date DATE,
10     job_id VARCHAR2,
11     salary NUMBER,
12    commission_pct NUMBER,
13    manager_id NUMBER,
14    department_id NUMBER
15  ) IS
16  BEGIN
17      INSERT INTO employees
18        VALUES (employee_id,
19                last_name,
20                first_name,
21                email,
22                phone_number,
23                hire_date,
24                job_id,
25                salary,
26                commission_pct,
27                manager_id,
28                department_id);
29  END hire_employee;
30
31  -- Code for procedure fire_employee:
Main Features of PL/SQL

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;
/

Package body created.

SQL>

To invoke a packaged subprogram, you must know only name of the package and the name and parameters of the subprogram (therefore, you can change the implementation details inside the package body without affecting the invoking applications).

Example 1–20 invokes the emp_actions package procedures hire_employee and fire_employee.

Example 1–20  Invoking a Procedure in a Package

SQL> CALL emp_actions.hire_employee (300, 'Belden', 'Enrique',
  2    'EBELDEN', '555.111.2222',
  3    '31-AUG-04', 'AC_MGR', 9000,
  4    .1, 101, 110);

Call completed.

SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE
  3      ('Number of employees with higher salary: ' ||
  4      TO_CHAR(emp_actions.num_above_salary(120)));
  5
  6    emp_actions.fire_employee(300);
  7    END;
  8  /

Number of employees with higher salary: 34

PL/SQL procedure successfully completed.

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

**Conditional Compilation**

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

- Use 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.

For more information, see Conditional Compilation on page 2-48.

**Embedded SQL Statements**

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 1–21.

**Example 1–21  Processing Query Results in a LOOP**

```
SQL> BEGIN
2    FOR someone IN (SELECT * FROM employees WHERE employee_id < 120)
3    LOOP
4      DBMS_OUTPUT.PUT_LINE('First name = ' || someone.first_name ||
5                           ', Last name = ' || someone.last_name);
6    END LOOP;
7  END;
8  /
First name = Steven, Last name = King
First name = Neena, Last name = Kochhar
First name = Lex, Last name = De Haan
First name = Alexander, Last name = Hunold
First name = Bruce, Last name = Ernst
First name = David, Last name = Austin
First name = Valli, Last name = Pataballa
First name = Diana, Last name = Lorentz
First name = Nancy, Last name = Greenberg
First name = Daniel, Last name = Faviet
First name = John, Last name = Chen
First name = Ismael, Last name = Sciarra
First name = Jose Manuel, Last name = Urman
First name = Luis, Last name = Popp
First name = Den, Last name = Raphaely
First name = Alexander, Last name = Khoo
First name = Shelli, Last name = Baida
First name = Sigal, Last name = Tobias
First name = Guy, Last name = Himuro
First name = Karen, Last name = Colmenares
```
PL/SQL procedure successfully completed.

SQL>

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.

Architecture of PL/SQL

Topics:

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

PL/SQL Engine

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

In either environment, the PL/SQL engine accepts as input any valid PL/SQL unit. The engine executes procedural statements, but sends SQL statements to the SQL engine in the database, as shown in Figure 1–2.

Figure 1–2 PL/SQL Engine

Typically, the database processes PL/SQL units.

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

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

A PL/SQL unit is any one of the following:

- PL/SQL block
- FUNCTION
- PACKAGE
- PACKAGE BODY
- PROCEDURE
- TRIGGER
- TYPE
- TYPE BODY

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

Table 1–1 lists and briefly describes the PL/SQL compilation parameters. For more information about these parameters, see Oracle Database Reference.

To display the values of these parameters, use the static data dictionary view ALL_PLSQL_OBJECT_SETTINGS. For more information about this view, see Oracle Database Reference.

Table 1–1  PL/SQL Compilation Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLSCOPE SETTINGS(^1)</td>
<td>Controls the compile-time collection, cross reference, and storage of PL/SQL source code identifier data. Used by the PL/Scope tool, which is described in Oracle Database Advanced Application Developer's Guide.</td>
</tr>
<tr>
<td>PLSQL_CCFLAGS (^1)</td>
<td>Enables you to control conditional compilation of each PL/SQL unit independently.</td>
</tr>
<tr>
<td>PLSQL_CODE_TYPE (^1)</td>
<td>Specifies the compilation mode for PL/SQL units—INTERPRETED (the default) or NATIVE. If the optimization level (set by PLSQL_OPTIMIZE_LEVEL) is less than 2:</td>
</tr>
<tr>
<td>PLSQL_DEBUG (^1)</td>
<td>Specifies whether or not PL/SQL units will be compiled for debugging. See note following table.</td>
</tr>
<tr>
<td>PLSQL_NATIVE_LIBRARY_DIR</td>
<td>Has no effect. See note following table.</td>
</tr>
<tr>
<td>PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT</td>
<td>Has no effect. See note following table.</td>
</tr>
</tbody>
</table>
The compile-time values of most of the parameters in Table 1–1 are stored with the metadata of the PL/SQL unit, which means you can reuse those values 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 COMPIL...`
   - `ALTER PACKAGE COMPIL...`
   - `ALTER PROCEDURE COMPIL...`

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 COMPIL...` statement, then the value of the compilation parameter is its value for the session.

---

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| `PLSQL_OPTIMIZE_LEVEL` | Specifies the optimization level at which to compile PL/SQL units (the higher the level, the more optimizations the compiler tries to make).  
If `PLSQL_OPTIMIZE_LEVEL=1`, PL/SQL units will be compiled for debugging. |
| `PLSQL_WARNINGS` | 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` | Enables you to create `CHAR` and `VARCHAR2` 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 unit.

---

**Note:** The following compilation parameters are deprecated and might be unavailable in future Oracle Database releases:

- `PLSQL_DEBUG`  
  For Release 11.1, it has the same effect as it had for Release 10.2—described in Table 1–1—but the compiler warns you that it is deprecated.  
  Instead of `PLSQL_DEBUG`, Oracle recommends using `PLSQL_OPTIMIZE_LEVEL=1`

- `PLSQL_NATIVE_LIBRARY_DIR`  
  For Release 11.1, it has no effect. The compiler does not warn you that it is deprecated.

- `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT`  
  For Release 11.1, it has no effect. The compiler does not warn you that it is deprecated.
See Also:

- ALTER FUNCTION Statement on page 14-3
- ALTER PACKAGE Statement on page 14-6
- ALTER PROCEDURE Statement on page 14-9
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
- Using PL/SQL to Create 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 Data Types 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

You must separate adjacent identifiers by a space or punctuation. For example:

```
SQL> BEGIN
  2  IF x > y THEN high := x; END IF; -- correct
  3  IF x > y THEN high := x; ENDF;  -- incorrect
  4  END;
  5  /
END;
```

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

```
SQL>
```

You cannot embed spaces inside lexical units (except string literals and comments). For example:

```
SQL> BEGIN
  2  count := count + 1; -- correct
  3  count := count + 1; -- incorrect
  4  END;
  5  /
      count := count + 1; -- incorrect
```

ERROR at line 3:
ORA-06550: line 3, column 9:
PLS-00103: Encountered the symbol ":" when expecting one of the following:
  := . ( @ % ;

```
SQL>
```

To show structure, you can split lines using carriage returns, and indent lines using spaces or tabs. For example:

```
SQL> DECLARE
  2    x    NUMBER := 10;
  3    y    NUMBER := 5;
  4    max  NUMBER;
  5  BEGIN
  6    IF x>y THEN max:=x;ELSE max:=y;END IF; -- correct but hard to read
     7
     8        -- Easier to read:
  9
 10    IF x > y THEN
 11      max:=x;
 12    ELSE
 13      max:=y;
 14    END IF;
 15  END;
 16 /
```

PL/SQL procedure successfully completed.

```
SQL>
```

Topics:

- Delimiters
Delimiters

A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. Table 2–1 lists the PL/SQL delimiters.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition operator</td>
</tr>
<tr>
<td>%</td>
<td>attribute indicator</td>
</tr>
<tr>
<td>'</td>
<td>character string delimiter</td>
</tr>
<tr>
<td>.</td>
<td>component selector</td>
</tr>
<tr>
<td>/</td>
<td>division operator</td>
</tr>
<tr>
<td>(</td>
<td>expression or list delimiter</td>
</tr>
<tr>
<td>)</td>
<td>expression or list delimiter</td>
</tr>
<tr>
<td>:</td>
<td>host variable indicator</td>
</tr>
<tr>
<td>,</td>
<td>item separator</td>
</tr>
<tr>
<td>*</td>
<td>multiplication operator</td>
</tr>
<tr>
<td>&quot;</td>
<td>quoted identifier delimiter</td>
</tr>
<tr>
<td>=</td>
<td>relational operator</td>
</tr>
<tr>
<td>&lt;</td>
<td>relational operator</td>
</tr>
<tr>
<td>&gt;</td>
<td>relational operator</td>
</tr>
<tr>
<td>@</td>
<td>remote access indicator</td>
</tr>
<tr>
<td>;</td>
<td>statement terminator</td>
</tr>
<tr>
<td>-</td>
<td>subtraction/negation operator</td>
</tr>
<tr>
<td>:=</td>
<td>assignment operator</td>
</tr>
<tr>
<td>=&gt;</td>
<td>association operator</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>exponentiation operator</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>label delimiter (begin)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>label delimiter (end)</td>
</tr>
<tr>
<td>/*</td>
<td>multi-line comment delimiter (begin)</td>
</tr>
<tr>
<td>*/</td>
<td>multi-line comment delimiter (end)</td>
</tr>
<tr>
<td>..</td>
<td>range operator</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>relational operator</td>
</tr>
<tr>
<td>!=</td>
<td>relational operator</td>
</tr>
<tr>
<td>~=</td>
<td>relational operator</td>
</tr>
<tr>
<td>^=</td>
<td>relational operator</td>
</tr>
</tbody>
</table>
Identifiers

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

The minimum length of an identifier is one character; the maximum length is 30 characters. The first character must be a letter, but each later character can be either a letter, numeral, dollar sign ($), underscore (_), or number sign (#). For example, the following are acceptable identifiers:

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

Characters other than the aforementioned are not allowed in identifiers. For example, the following are not acceptable identifiers:

mine&yours  -- ampersand (&) is not allowed
debit-amount -- hyphen (-) is not allowed
on/off       -- slash (/) is not allowed
user id      -- space is not allowed

PL/SQL is not case-sensitive with respect to identifiers. For example, PL/SQL considers the following to be the same:

lastname
LastName
LASTNAME

Every character, alphabetic or not, is significant. For example, PL/SQL considers the following to be different:

lastname
last name

Make your identifiers meaningful rather than obscure. For example, the meaning of cost_per_thousand is obvious, while the meaning of cpt is not.

Topics:

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

Both reserved words and keywords have special meaning in 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.

Trying to redefine a reserved word causes a compilation error. For example:

```
SQL> DECLARE
2  end BOOLEAN;
3  BEGIN
4  NULL;
5  END;
6  /
   end BOOLEAN;
   *
```

ERROR at line 2:
ORA-06550: line 2, column 3:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
begin function pragma procedure subtype type <an identifier>
<a double-quoted delimited-identifier> current cursor delete
exists prior
The symbol 'begin was inserted before "END" to continue.
ORA-06550: line 5, column 4:
PLS-00103: Encountered the symbol "end-of-file" when expecting one of the
following:
{ begin case declare end exception exit for goto if loop mod
null pragma raise return select update while with
<an identifier> <a double-quoted

SQL>
```

The PL/SQL reserved words are listed in Table D–1 on page D-1.

Keywords also have special meaning in PL/SQL, but you can redefine them (this is not recommended). The PL/SQL keywords are listed in 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 about the PL/SQL data types, see Predefined PL/SQL Scalar Data Types 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 data type; otherwise this literal has the NUMBER data type. 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_DOUBLE respectively. For the properties of the data types, see Predefined PL/SQL Numeric Data Types 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 preceding 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 about results outside the valid range, see NUMBER Data Type on page 3-6.
Example 2–1  NUMBER Literals

```sql
SQL> DECLARE
2  n NUMBER;
3  BEGIN
4  n := -9.999999E-130;
5  n := 9.999E125;
6  n := 10.0E125;
7  END;
8  /
9  n := 10.0E125;
* ERROR at line 6:
ORA-06550: line 6, column 8:
PLS-00569: numeric overflow or underflow
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
SQL>
```

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

```sql
SQL> DECLARE
2  x BINARY_FLOAT := sqrt(2.0f);
3  y BINARY_DOUBLE := sqrt(2.0d);
4  BEGIN
5  NULL;
6  END;
7  /
PL/SQL procedure successfully completed.
SQL>
```

Character Literals

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

```
'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 data type CHAR. For example:

```
'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 need not 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 data type 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 data type, as in Example 2–3.

**Example 2–3  Using DateTime Literals**

```
SQL> DECLARE
2   d1 DATE := DATE '1998-12-25';
3   t1 TIMESTAMP := TIMESTAMP '1997-10-22 13:01:01';
4
5   t2 TIMESTAMP WITH TIME ZONE :=
6       TIMESTAMP '1997-01-31 09:26:56.66 +02:00';
7
8   -- Three years and two months
9   -- For greater precision, use the day-to-second interval
10
11   i1 INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO MONTH;
12
13   -- Five days, four hours, three minutes, two and 1/100 seconds
14```
15     i2 INTERVAL DAY TO SECOND :=
16          INTERVAL '5 04:03:02.01' DAY TO SECOND;
17
18 BEGIN
19      NULL;
20 END;
21 /
PL/SQL procedure successfully completed.

SQL>

See Also:
- Oracle Database SQL Language Reference for syntax of date and time types
- Oracle Database Advanced Application Developer’s Guide for examples of date and time arithmetic

Comments

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

Topics:
- Single-Line Comments
- Multiline Comments

See Also: Comment on page 13-31

Single-Line Comments

A single-line comment begins with --. It can appear anywhere on a line, and it extends to the end of the line, as in Example 2–4.

Example 2–4  Single-Line Comments

SQL> DECLARE
2     howmany    NUMBER;
3     num_tables NUMBER;
4    BEGIN
5      -- Begin processing
6      SELECT COUNT(*) INTO howmany
7        FROM USER_OBJECTS
8      WHERE OBJECT_TYPE = 'TABLE'; -- Check number of tables
9      num_tables := howmany;        -- Compute some other value
10    END;
11 /
PL/SQL procedure successfully completed.

SQL>

While testing or debugging a program, you might want to disable a line of code by making it a comment. For example:
Multiline Comments

A multiline comments begins with /*, ends with */ , and can span multiple lines, as in Example 2–5. You can use multiline comment delimiters to "comment out" sections of code.

**Example 2–5  Multiline Comment**

```sql
SQL> DECLARE
2     some_condition  BOOLEAN;
3     pi              NUMBER := 3.1415926;
4     radius          NUMBER := 15;
5     area            NUMBER;
6  BEGIN
7  /* Perform some simple tests and assignments */
8    IF 2 + 2 = 4 THEN
9      some_condition := TRUE;
10     /* We expect this THEN to always be performed */
11    END IF;
12  /* The following line computes the area of a circle using pi, 
13    which is the ratio between the circumference and diameter. 
14    After the area is computed, the result is displayed. */
15    area := pi * radius**2;
16    DBMS_OUTPUT.PUT_LINE('The area is: ' || TO_CHAR(area));
17  END;
18  /
The area is: 706.858335
PL/SQL procedure successfully completed.
SQL>
```

**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 data type, and name the storage location so that you can reference it.

Topics:
- Variables
- Constants
- Using DEFAULT
- Using NOT NULL
- Using the %TYPE Attribute
- Using the %ROWTYPE Attribute
- Restrictions on Declarations
Variables

Example 2–6 declares a variable of type `DATE`, a variable of type `SMALLINT` (to which it assigns the initial value zero), and three variables of type `REAL`. The expression following the assignment operator can be arbitrarily complex, and can refer to previously initialized variables, as in the declaration of the variable `area`.

Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL.

Example 2–6  Declaring Variables

```sql
SQL> DECLARE
2    birthday   DATE;
3    emp_count  SMALLINT := 0;
4    pi         REAL := 3.14159;
5    radius     REAL := 1;
6    area       REAL := pi * radius**2;
7  BEGIN
8    NULL;
9  END;
10  /
```

PL/SQL procedure successfully completed.

SQL>

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. Constants are initialized every time a block or subprogram is entered.

Example 2–7  Declaring Constants

```sql
SQL> DECLARE
2    credit_limit      CONSTANT REAL    := 5000.00;
3    max_days_in_year  CONSTANT INTEGER := 366;
4    urban_legend      CONSTANT BOOLEAN := FALSE;
5  BEGIN
6    NULL;
7  END;
8  /
```

PL/SQL procedure successfully completed.

SQL>

Using DEFAULT

You can use the keyword `DEFAULT` instead of the assignment operator to initialize variables. You can also use `DEFAULT` to initialize subprogram parameters, cursor parameters, and fields in a user-defined record.

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.
Example 2–8 Assigning Default Values to Variables with DEFAULT Keyword

```sql
DECLARE
  blood_type CHAR DEFAULT 'O'; -- Same as blood_type CHAR := 'O';
  hours_worked INTEGER DEFAULT 40; -- Typical value
  employee_count INTEGER := 0; -- No typical value
BEGIN
  NULL;
END;
/]
```

PL/SQL procedure successfully completed.

Using NOT NULL

A declaration can impose the NOT NULL constraint, which prevents you from assigning a null value to the variable. Because variables are initialized to NULL by default, a declaration that specifies NOT NULL must also specify a default value.

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

Example 2–9 Declaring Variables with NOT NULL Constraint

```sql
DECLARE
  acct_id INTEGER(4) NOT NULL := 9999;
  a NATURALN := 9999;
  b POSITIVEN := 9999;
  c SIMPLE_INTEGER := 9999;
BEGIN
  NULL;
END;
/]
```

PL/SQL procedure successfully completed.

Using the %TYPE Attribute

The %TYPE attribute lets you declare a constant, variable, field, or parameter to be of the same data type a previously declared variable, field, record, nested table, or database column. If the referenced item changes, your declaration is automatically updated. You need not change your code when, for example, the length of a VARCHAR2 column increases.

An item declared with %TYPE (the referencing item) always inherits the data type of the referenced item. The referencing item inherits the constraints only if the referenced item is not a database column. The referencing item inherits the default value only if the referencing item is not a database column and does not have the NOT NULL constraint.

In Example 2–10, the variable debit inherits the data type of the variable credit. The variables upper_name, lower_name, and init_name inherit the data type and default value of the variable name.
Example 2–10 Using %TYPE to Declare Variables of the Types of Other Variables

```sql
SQL> DECLARE
2    credit  PLS_INTEGER RANGE 1000..25000;
3    debit  credit%TYPE; -- inherits data type
4
5    name        VARCHAR2(20) := 'JoHn SmItH';
6    upper_name  name%TYPE; -- inherits data type and default value
7    lower_name  name%TYPE; -- inherits data type and default value
8    init_name   name%TYPE; -- inherits data type and default value
9 BEGIN
10    DBMS_OUTPUT.PUT_LINE ('name: ' || name);
11    DBMS_OUTPUT.PUT_LINE ('upper_name: ' || UPPER(name));
12    DBMS_OUTPUT.PUT_LINE ('lower_name: ' || LOWER(name));
13    DBMS_OUTPUT.PUT_LINE ('init_name:  ' || INITCAP(name));
14 END;
15 /
name: JoHn SmItH
upper_name: JOHN SMITH
lower_name: john smith
init_name:  John Smith

PL/SQL procedure successfully completed.
```

If you add a NOT NULL constraint to the variable name in Example 2–10, and declare another variable that references it, you must specify a default value for the new item, as Example 2–11 shows.

Example 2–11 Using %TYPE Incorrectly with NOT NULL Referenced Type

```sql
SQL> DECLARE
2    name    VARCHAR2(20) NOT NULL := 'JoHn SmItH';
3    same_name   name%TYPE;
4 BEGIN
5    NULL;
6 END;
7 /
same_name   name%TYPE;
```

*ERROR at line 3:
ORA-06550: line 3, column 15:
PLS-00218: a variable declared NOT NULL must have an initialization assignment
```

In Example 2–12, the variables upper_name, lower_name, and init_name inherit the data type and NOT NULL constraint of the variable name, but not its default value. To avoid the error shown in Example 2–11, they are assigned their own default values.

Example 2–12 Using %TYPE Correctly with NOT NULL Referenced Type

```sql
SQL> DECLARE
2    name        VARCHAR2(20) NOT NULL := 'JoHn SmItH';
3    upper_name  name%TYPE := UPPER(name);
4    lower_name  name%TYPE := LOWER(name);
5    init_name   name%TYPE := INITCAP(name);
6 BEGIN
7    DBMS_OUTPUT.PUT_LINE ('name: ' || name);
8    DBMS_OUTPUT.PUT_LINE ('upper_name: ' || upper_name);
```

In Example 2–10, the variables credit and debit are declared using %TYPE. The variable name is declared as VARCHAR2(20) and assigned a default value. The variables upper_name, lower_name, and init_name inherit the data type and NOT NULL constraint of the variable name, but not its default value. To avoid the error shown in Example 2–11, they are assigned their own default values.
The %TYPE attribute is particularly useful when declaring variables that refer to database columns. When you use `table_name.column_name.%TYPE` to declare a data item, you need not know the referenced data type or its attributes (such as precision, scale, and length), and if they change, you need not update your code.

Example 2–13 shows that referencing items do not inherit column constraints or default values from database columns.

**Example 2–13  Using %TYPE to Declare Variables of the Types of Table Columns**

```sql
SQL> CREATE TABLE employees_temp ( 2   empid NUMBER(6) NOT NULL PRIMARY KEY, 3   deptid NUMBER(6) CONSTRAINT c_employees_temp_deptid 4     CHECK (deptid BETWEEN 100 AND 200), 5   deptname VARCHAR2(30) DEFAULT 'Sales' 6 ) ;
Table created.
```

```sql
SQL> DECLARE 2    v_empid employees_temp.empid%TYPE; 3    v_deptid employees_temp.deptid%TYPE; 4    v_deptname employees_temp.deptname%TYPE; 5  BEGIN 6    v_empid := NULL; -- Null constraint not inherited 7    v_deptid := 50; -- Check constraint not inherited 8    DBMS_OUTPUT.PUT_LINE 9      ('v_deptname: ' || v_deptname); -- Default value not inherited 10  END; 11  /
```

PL/SQL procedure successfully completed.

**See Also:**
- Constraints and Default Values with Subtypes on page 3-26 for information about column constraints that are inherited by subtypes declared using %TYPE
- %TYPE Attribute on page 13-128 for the syntax of the %TYPE attribute
Using the %ROWTYPE Attribute

The %ROWTYPE attribute lets you declare a record that represents a row in a table or view. For each column in the referenced table or view, the record has a field with the same name and data type. To reference a field in the record, use record_.field_name. The record fields do not inherit the constraints or default values of the corresponding columns, as Example 2–14 shows.

If the referenced item table or view changes, your declaration is automatically updated. You need not change your code when, for example, columns are added or dropped from the table or view.

Example 2–14 Using %ROWTYPE to Declare a Record that Represents a Table Row

```
SQL> CREATE TABLE employees_temp (
  2    empid  NUMBER(6) NOT NULL PRIMARY KEY,
  3    deptid NUMBER(6) CONSTRAINT c_employees_temp_deptid
  4      CHECK (deptid BETWEEN 100 AND 200),
  5    deptname VARCHAR2(30) DEFAULT 'Sales'
  6  );
Table created.
SQL>
SQL> DECLARE
  2  emprec  employees_temp%ROWTYPE;
  3  BEGIN
  4    emprec.empid := NULL; -- Null constraint not inherited
  5    emprec.deptid := 50; -- Check constraint not inherited
  6    DBMS_OUTPUT.PUT_LINE ('emprec.deptname: ' || emprec.deptname);
  7    -- Default value not inherited
  8  END;
  9  /
emprec.deptname:
PL/SQL procedure successfully completed.
SQL>
```

The record emprec in Example 2–14 has a field for every column in the table employees_temp. The record dept_rec in Example 2–15 has columns for a subset of columns in the departments table.

Example 2–15 Declaring a Record that Represents a Subset of Table Columns

```
SQL> DECLARE
  2    CURSOR c1 IS
  3      SELECT department_id, department_name
  4        FROM departments;
  5    dept_rec c1%ROWTYPE; -- includes subset of columns in table
  6  BEGIN
  7    NULL;
  8  END;
  9  /
emprec.deptname:
PL/SQL procedure successfully completed.
SQL>
See Also:  Example 3–15 on page 3-27
```
PL/SQL procedure successfully completed.

SQL>

The record join_rec in Example 2–15 has columns from two tables, employees and departments.

Example 2–16 Declaring a Record that Represents a Row from a Join

```
SQL> DECLARE
2    CURSOR c2 IS
3      SELECT employee_id, email, employees.manager_id, location_id
4        FROM employees, departments
5          WHERE employees.department_id = departments.department_id;
6
7    join_rec  c2%ROWTYPE; -- includes columns from two tables
8
9  BEGIN
10    NULL;
11  END;
12 /
```

PL/SQL procedure successfully completed.

SQL>

Topics:

- Aggregate Assignment
- Using Aliases

Aggregate Assignment

A %ROWTYPE declaration cannot include an initialization clause, but there are two ways to assign values to all fields of a record at once:

- If their declarations refer to the same table or cursor, you can assign one record to another, as in Example 2–17.
- Use the SELECT or FETCH statement to assign a list of column values to a record. The column names must appear in the order in which they were defined in the CREATE TABLE or CREATE VIEW statement that created the referenced table or view. 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.

Example 2–17 Assigning One Record to Another, Correctly and Incorrectly

```
SQL> DECLARE
2    dept_rec1  departments%ROWTYPE;
3    dept_rec2  departments%ROWTYPE;
4
5    CURSOR c1 IS SELECT department_id, location_id
6      FROM departments;
7
8    dept_rec3  c1%ROWTYPE;
9    dept_rec4  c1%ROWTYPE;
10
11  BEGIN
12    dept_rec1 := dept_rec2; -- declarations refer to same table
13    dept_rec3 := dept_rec4; -- declarations refer to same cursor
14 /
```

PL/SQL procedure successfully completed.
Example 2–18 uses the SELECT INTO statement to assign a list of column values to a record.

**Example 2–18 Using SELECT INTO for Aggregate Assignment**

```sql
SQL> DECLARE
2    dept_rec departments%ROWTYPE;
3  BEGIN
4    SELECT * INTO dept_rec
5      FROM departments
6        WHERE department_id = 30
7          AND ROWNUM < 2;
8  END;
9  /
```

PL/SQL procedure successfully completed.

**Using Aliases**

Select-list items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases, such as complete_name in Example 2–19.

**Example 2–19 Using an Alias for an Expression Associated with %ROWTYPE**

```sql
SQL> BEGIN
2    FOR item IN
3      (SELECT (first_name || ' ' || last_name) complete_name
4        FROM employees
5           WHERE ROWNUM < 11
6      ) LOOP
7        DBMS_OUTPUT.PUT_LINE
8          ('Employee name: ' || item.complete_name);
9      END LOOP;
10  END;
11  /
```

Employee name: Ellen Abel
Employee name: Sundar Ande
Employee name: Mozhe Atkinson
Employee name: David Austin
Employee name: Hermann Baer
Employee name: Shelli Baida
Employee name: Amit Banda
Employee name: Elizabeth Bates
Employee name: Sarah Bell
Employee name: David Bernstein

PL/SQL procedure successfully completed.

SQL>

**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 Creating Nested Subprograms that Invoke Each Other on page 8-5.

Some languages enable you to declare a list of variables that have the same data type. PL/SQL does not allow this. You must declare each variable separately. To save space, you can put more than one declaration on a line. For example:

```sql
SQL> DECLARE
2  i, j, k, l SMALLINT;
3  BEGIN
4    NULL;
5  END;
6  /

i, j, k, l SMALLINT;
*
```

ERROR at line 2:
ORA-06550: line 2, column 4:
PLS-00103: Encountered the symbol "," when expecting one of the following:
constant exception <an identifier>
<a double-quoted delimited-identifier> table long double ref
char time timestamp interval date binary national character
nchar
ORA-06550: line 2, column 14:
PLS-00103: Encountered the symbol 'SMALLINT' when expecting one of the following:
. ( ) , * & % = - + < > at in is mod remainder not rem =>
<an exponent (**)> <> or != or ^= => and or like like2
like4 likec between ||
ORA-06550: line 5, column 4:
PLS-00103: Encountered the symbol 'end-of-file' when expecting one of the following:
( begin case declare end exception exit for goto if loop mod null pragma raise return select update while with
<an identifier> <a double-quoted

SQL> DECLARE
2  i SMALLINT; j SMALLINT; k SMALLINT; l SMALLINT;
3  BEGIN
4    NULL;
5  END;
6  /

PL/SQL procedure successfully completed.

SQL>
Naming Conventions

The same naming conventions apply to PL/SQL constants, variables, cursors, cursor variables, exceptions, procedures, functions, and packages. Names can be simple, qualified, remote, or both qualified and remote. For example:

- **Simple**—procedure name only:
  
  ```plsql```
  ```
  raise_salary(employee_id, amount);
  ```
  ```
  ```

- **Qualified**—procedure name preceded by the name of the package that contains it (this is called **dot notation** because a dot separates the package name from the procedure name):
  
  ```plsql```
  ```
  emp_actions.raise_salary(employee_id, amount);
  ```
  ```

- **Remote**—procedure name followed by the remote access indicator (@) and a link to the database on which the procedure is stored:
  
  ```plsql```
  ```
  raise_salary@newyork(employee_id, amount);
  ```
  ```

- **Qualified and remote**:
  
  ```plsql```
  ```
  emp_actions.raise_salary@newyork(employee_id, amount);
  ```
  ```

Topics:

- Scope
- Case Sensitivity
- Name Resolution
- Synonyms

Scope

Within the same scope, all declared identifiers must be unique. Even if their data types differ, variables and parameters cannot share the same name. An error occurs when the duplicate identifier is referenced, as in **Example 2–20**.

**Example 2–20  Duplicate Identifiers in Same Scope**

```sql```
```sql
SQL> DECLARE
2    id  BOOLEAN;
3    id  VARCHAR2(5);  -- duplicate identifier
4  BEGIN
5    id := FALSE;
6  END;
7  /
8  id := FALSE;
9  ```
10 ```
ERROR at line 5:
ORA-06550: line 5, column 3:
PLS-00371: at most one declaration for 'ID' is permitted
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored
```

For the scoping rules that apply to identifiers, see **Scope and Visibility of PL/SQL Identifiers** on page 2-22.
Case Sensitivity

Like all identifiers, the names of constants, variables, and parameters are not case sensitive, as Example 2–21 shows.

Example 2–21 Case Insensitivity of Identifiers

```
SQL> DECLARE
2  zip_code INTEGER;
3  Zip_Code INTEGER;
4  BEGIN
5    zip_code := 90120;
6  END;
7  /
8  zip_code := 90120;
9  *
```

ERROR at line 5:
ORA-06550: line 5, column 3:
PLS-00371: at most one declaration for 'ZIP_CODE' is permitted
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored

SQL>

Name Resolution

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

Caution: When a variable name is interpreted as a column name, data can be deleted unintentionally, as Example 2–22 shows. Example 2–22 also shows two ways to avoid this error.

Example 2–22 Using a Block Label for Name Resolution

```
SQL> CREATE TABLE employees2 AS
2    SELECT last_name FROM employees;
Table created.
SQL>
```

```
-- Deletes everyone, because both LAST_NAMES refer to the column:
SQL> BEGIN
2    DELETE FROM employees2
3      WHERE last_name = last_name
4    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
5  END;
6  /
```

Deleted 107 rows.

PL/SQL procedure successfully completed.

SQL> ROLLBACK;

Rollback complete.
SQL> -- Avoid error by giving column and variable different names:
SQL> DECLARE
2    last_name    VARCHAR2(10) := 'King';
3    v_last_name  VARCHAR2(10) := 'King';
4  BEGIN
5    DELETE FROM employees2
6      WHERE last_name = v_last_name;
7    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
8  END;
9  /
Deleted 2 rows.
PL/SQL procedure successfully completed.

SQL> ROLLBACK;
Rollback complete.

SQL> -- Avoid error by qualifying variable with block name:
SQL> <<main>> -- Label block for future reference
SQL> DECLARE
2    last_name    VARCHAR2(10) := 'King';
3    v_last_name  VARCHAR2(10) := 'King';
4  BEGIN
5    DELETE FROM employees2
6      WHERE last_name = main.last_name;
7    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
8  END;
9  /
Deleted 2 rows.
PL/SQL procedure successfully completed.

SQL> ROLLBACK;
Rollback complete.

You can use a subprogram name to qualify references to local variables and formal parameters, as in Example 2–23.

Example 2–23 Using a Subprogram Name for Name Resolution

SQL> DECLARE
2    FUNCTION dept_name (department_id IN NUMBER)
3      RETURN departments.department_name%TYPE
4    IS
5      department_name  departments.department_name%TYPE;
6  BEGIN
7    SELECT department_name INTO dept_name.department_name
8      -- column
9      FROM departments
10      WHERE department_id = dept_name.department_id;
11    RETURN department_name;
Scope and Visibility of PL/SQL Identifiers

You can use the SQL statement `CREATE SYNONYM` to create synonyms to provide location transparency for remote schema objects. You cannot create synonyms for items declared within PL/SQL subprograms or packages.

See: Oracle Database SQL Language Reference for information about the SQL statement `CREATE SYNONYM`

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 the region of a PL/SQL unit from which you can reference the identifier. The visibility of an identifier is the region of a PL/SQL unit from which you can reference the identifier without qualifying it.
An identifier declared in a PL/SQL unit is **local** to that unit and **global** to its subunits. If a subunit redeclares a global identifier, then inside the subunit, both identifiers are in scope, but only the local identifier is visible. To reference the global identifier, the subunit must qualify it.

You cannot declare an identifier twice in the same PL/SQL unit, but you can declare the same identifier in two different units. The two items represented by the identifier are distinct, and changing one does not affect the other.

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

**Example 2–24** shows the scope and visibility of several global and local identifiers. The global identifier `a` is redeclared in the first sub-block.

**Example 2–24  Scope and Visibility of Identifiers**

```sql
SQL> DECLARE
2  a CHAR; -- Scope of a (CHAR) begins
3  b REAL; -- Scope of b begins
4  BEGIN
5    -- Visible: a (CHAR), b
6
7    DECLARE
8      a INTEGER; -- Scope of a (INTEGER) begins
9      c REAL; -- Scope of c begins
10    BEGIN
11      -- Visible: a (INTEGER), b, c
12      NULL;
13    END; -- Scopes of a (INTEGER) and c end
14
15    DECLARE
16      d REAL; -- Scope of d begins
17    BEGIN
18      -- Visible: a (CHAR), b, d
19      NULL;
20    END; -- Scope of d ends
21
22    -- Visible: a (CHAR), b
23  END; -- Scopes of a (CHAR) and b end
24 /

PL/SQL procedure successfully completed.
```

**Example 2–25** declares the variable `birthdate` in a labeled block, `outer`, redeclares it in a sub-block, and then references it in the sub-block by qualifying its name with the block label.

**Example 2–25  Qualifying a Redeclared Global Identifier with a Block Label**

```sql
SQL> <<outer>>
2  DECLARE
3    birthdate DATE := '09-AUG-70';
4  BEGIN
5    DECLARE
6      birthdate DATE;
7    BEGIN
8      birthdate := '29-SEP-70';
9  
```
Example 2–26 declares the variable rating in a procedure, check_credit, redeclares it in a function within the procedure, and then references it in the function by qualifying its name with the procedure name. (The built-in SQL function TO_CHAR returns the character equivalent of its argument. For more information about TO_CHAR, see Oracle Database SQL Language Reference.)

Example 2–26  Qualifying an Identifier with a Subprogram Name

```sql
SQL> CREATE OR REPLACE PROCEDURE check_credit (limit NUMBER) AS
    2    rating NUMBER := 3;
    3    
    4    FUNCTION check_rating RETURN BOOLEAN IS
    5       rating NUMBER := 1;
    6       over_limit BOOLEAN;
    7       BEGIN
    8       IF check_credit.rating <= limit THEN
    9          over_limit := FALSE;
   10       ELSE
   11          over_limit := TRUE;
   12          rating := limit;
   13       END IF;
   14       RETURN over_limit;
   15     END check_rating;
   16     BEGIN
   17     IF check_rating THEN
   18        DBMS_OUTPUT.PUT_LINE ('Credit rating over limit (' || TO_CHAR(limit) || '). ' || 'Rating: ' || TO_CHAR(rating));
   19     ELSE
   20        DBMS_OUTPUT.PUT_LINE ('Credit rating OK. ' || 'Rating: ' || TO_CHAR(rating));
   21     END IF;
   22     END;
   23     
   24     END;
   25     
   26 /
```

Procedure created.

SQL> BEGIN
    2    check_credit(1);
    3    END;
    4    /
```
Credit rating over limit (1).  Rating: 3

PL/SQL procedure successfully completed.

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

Example 2–27 has both a block and a subprogram named `echo`. Both the block and the subprogram declare a variable named `x`. Within the subprogram, `echo.x` refers to the local variable `x`, not to the global variable `x`.

Example 2–27  Label and Subprogram with Same Name in Same Scope

```sql
SQL> <<echo>>
2  DECLARE
3    x  NUMBER := 5;
4
5    PROCEDURE echo AS
6      x  NUMBER := 0;
7    BEGIN
8      DBMS_OUTPUT.PUT_LINE('x = ' || x);
9      DBMS_OUTPUT.PUT_LINE('echo.x = ' || echo.x);
10    END;
11
12  BEGIN
13    echo;
14  END;
15 /
16 x = 0
17 echo.x = 0

PL/SQL procedure successfully completed.
```

Example 2–28 has both a block and a subprogram named `echo`. Both the block and the subprogram declare a variable named `x`. Within the subprogram, `echo.x` refers to the local variable `x`, not to the global variable `x`.

Example 2–28 has two labels for the outer block, `compute_ratio` and `another_label`. The second label is reused in the inner block. Within the inner block, `another_label.denominator` refers to the local variable denominator, not to the global variable denominator, which results in the error `ZERO_DIVIDE`.

Example 2–28  Block with Multiple and Duplicate Labels

```sql
SQL> <<compute_ratio>>
2  <<another_label>>
3  DECLARE
4    numerator  NUMBER := 22;
5    denominator NUMBER := 7;
6  BEGIN
7    <<another_label>>
8    DECLARE
9      denominator NUMBER := 0;
10   BEGIN
11      DBMS_OUTPUT.PUT_LINE('Ratio with compute_ratio.denominator = ');
12      DBMS_OUTPUT.PUT_LINE(numerator/compute_ratio.denominator);
13      DBMS_OUTPUT.PUT_LINE('Ratio with another_label.denominator = ');
14      DBMS_OUTPUT.PUT_LINE(numerator/another_label.denominator);
15    EXCEPTION
```
Assigning Values to Variables

You can assign a default value to a variable when you declare it (as explained in Variables on page 2-11) or after you have declared it, with an assignment statement. For example, the following statement assigns a new value to the variable `bonus`, overwriting its old value:

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

The expression following the assignment operator (:=) can be arbitrarily complex, but it must yield a data type that is the same as, or convertible to, the data type of the variable.

Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to `NULL`. Unless you explicitly initialize a variable, its value is `NULL`, as Example 2–29 shows.

**Example 2–29  Variable Initialized to NULL by Default**

```
SQL> DECLARE
2    counter INTEGER;
3  BEGIN
4     counter := counter + 1;
5
6     IF counter IS NULL THEN
7         DBMS_OUTPUT.PUT_LINE('counter is NULL.');
8     END IF;
9  END;
10  /
counter is NULL.
PL/SQL procedure successfully completed.
```

To avoid unexpected results, never reference a variable before assigning it a value.

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, either as literals or as the results of expressions.

In Example 2–30, the BOOLEAN variable done is initialized to NULL by default, assigned the literal value FALSE, compared to a literal BOOLEAN value, and assigned the value of a BOOLEAN expression.

*Example 2–30  Assigning BOOLEAN Values*

```sql
SQL> DECLARE
2    done    BOOLEAN;            -- Initialize to NULL by default
3    counter NUMBER := 0;
4  BEGIN
5    done := FALSE;              -- Assign literal value
6    WHILE done != TRUE          -- Compare to literal value
7    LOOP
8      counter := counter + 1;
9    done := (counter > 500);  -- Assign value of BOOLEAN expression
10    END LOOP;
11  END;
12  /
PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You can use the SELECT INTO statement to 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 in Example 2–31.

*Example 2–31  Assigning Query Results to Variables*

```sql
SQL> DECLARE
2    emp_id    employees.employee_id%TYPE := 100;
3    emp_name  employees.last_name%TYPE;
4    wages     NUMBER(7,2);
5  BEGIN
6    SELECT last_name, salary + (salary * nvl(commission_pct,0))
7    INTO emp_name, wages
8    FROM employees
9    WHERE employee_id = emp_id;
10
11    DBMS_OUTPUT.PUT_LINE
12      ('Employee ' || emp_name || ' might make ' || wages);
13  END;
14  /
Employee King might make 24000
PL/SQL procedure successfully completed.
```

Because SQL does not have a BOOLEAN type, you cannot select column values into a BOOLEAN variable. For more information about 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

The simplest PL/SQL expression consists of a single variable, which yields a value directly. You can build arbitrarily complex PL/SQL expressions from operands and operators. An operand is a variable, constant, literal, placeholder, or function call. An operator is either unary or binary, operating on either one operand or two operands, respectively. An example of a unary operator is negation (−). An example of a binary operator is addition (+).

An example of a simple arithmetic expression is:

\[-X \div 2 + 3\]

PL/SQL evaluates an expression by combining the values of the operands as specified by the operators. An expression always returns a single value. PL/SQL determines the data type of this value by examining the expression and the context in which it appears.

Topics:
- Concatenation Operator
- Operator Precedence
- Logical Operators
- BOOLEAN Expressions
- CASE Expressions
- Handling NULL Values in Comparisons and Conditional Statements

Concatenation Operator

The concatenation operator (||) appends one string operand to another. Each string can be CHAR, VARCHAR2, CLOB, or the equivalent Unicode-enabled type. If either string is a CLOB, the result is a temporary CLOB; otherwise, it is a VARCHAR2 value.

Example 2–32 and many other examples in this book use the concatenation operator.

**Example 2–32  Concatenation Operator**

```sql
SQL> DECLARE
2  x VARCHAR2(4) := 'suit';
3  y VARCHAR2(4) := 'case';
4  BEGIN
5    DBMS_OUTPUT.PUT_LINE (x || y);
6  END;
7  /
suitcase

PL/SQL procedure successfully completed.

SQL>
```

Operator Precedence

The operations within an expression are evaluated in order of precedence. Table 2–2 shows operator precedence from highest to lowest. Operators with equal precedence are applied in no particular order.
You can use parentheses to control the order of evaluation. When parentheses are nested, the most deeply nested subexpression is evaluated first. You can use parentheses to improve readability, even when you do not need them to control the order of evaluation. (In Example 2-33, the built-in SQL function TO_CHAR returns the character equivalent of its argument. For more information about TO_CHAR, see Oracle Database SQL Language Reference.)

Example 2–33 Operator Precedence

```sql
SQL> DECLARE
2  salary  NUMBER := 60000;
3  commission  NUMBER := 0.10;
4 BEGIN
5  -- Division has higher precedence than addition:
6  DBMS_OUTPUT.PUT_LINE('5 + 12 / 4 = ' || TO_CHAR(5 + 12 / 4));
7  DBMS_OUTPUT.PUT_LINE('12 / 4 + 5 = ' || TO_CHAR(12 / 4 + 5));
8
9  -- Parentheses override default operator precedence:
10 DBMS_OUTPUT.PUT_LINE('8 + 6 / 2 = ' || TO_CHAR((8 + 6) / 2));
11 DBMS_OUTPUT.PUT_LINE('(8 + 6) / 2 = ' || TO_CHAR((8 + 6) / 2));
12
13  -- Most deeply nested subexpression is evaluated first:
14 DBMS_OUTPUT.PUT_LINE('100 + (20 / 5 + (7 - 3)) = '
15  || TO_CHAR(100 + (20 / 5 + (7 - 3))));
16
17  -- Parentheses, even when unnecessary, improve readability:
18 DBMS_OUTPUT.PUT_LINE(''(salary * 0.05) + (commission * 0.25) = '  
19  || TO_CHAR((salary * 0.05) + (commission * 0.25)) )
20 );
21
22  DBMS_OUTPUT.PUT_LINE(''(salary * 0.05 + commission * 0.25 = '  
23  || TO_CHAR(salary * 0.05 + commission * 0.25) )
24 );
25
26  DBMS_OUTPUT.PUT_LINE('5 + 12 / 4 = 8
27 12 / 4 + 5 = 8
28 8 + 6 / 2 = 11
```
(8 + 6) / 2 = 7
100 + (20 / 5 + (7 - 3)) = 108
(salary * 0.05) + (commission * 0.25) = 3000.025
salary * 0.05 + commission * 0.25 = 3000.025

PL/SQL procedure successfully completed.

SQL>

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>
<thead>
<tr>
<th>Table 2–3 Logical Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>

Be careful to avoid unexpected results in expressions involving NULL. For more information, see Handling NULL Values in Comparisons and Conditional Statements on page 2-42.

As Table 2–3 and Example 2–34 show, AND returns TRUE if and only if both operands are TRUE. (Several examples use the print_boolean procedure that Example 2–34 creates.)

Example 2–34   AND Operator

SQL> CREATE OR REPLACE PROCEDURE print_boolean ( 
2  name VARCHAR2, 
3  value BOOLEAN 
4 ) IS 
5  BEGIN 
6  IF value IS NULL THEN 
7    DBMS_OUTPUT.PUT_LINE (name || ' = NULL'); 
8  ELSIF value = TRUE THEN 
9    DBMS_OUTPUT.PUT_LINE (name || ' = TRUE'); 
10  ELSE 
11    DBMS_OUTPUT.PUT_LINE (name || ' = FALSE'); 
12  END IF; 
13  END; 
14 /

Procedure created.

SQL> DECLARE

PL/SQL procedure successfully completed.
PL/SQL Expressions and Comparisons

As Table 2–3 and Example 2–35 show, OR returns TRUE if either operand is TRUE. (Example 2–35 invokes the print_boolean procedure created in Example 2–34.)
As Table 2–3 and Example 2–36 show, NOT returns the opposite of its operand, unless the operand is NULL. NOT NULL returns NULL, because NULL is an indeterminate value. (Example 2–36 invokes the print_boolean procedure created in Example 2–34.)

Example 2–36  NOT Operator

```
SQL> DECLARE
2  PROCEDURE print_not_x (x BOOLEAN
3     print_not_x (x BOOLEAN) IS
4       print_boolean ('x', x);
5       print_boolean ('y', y);
6       print_boolean ('x OR y', x OR y);
7     END;
8
9  BEGIN
10    print_x_or_y (FALSE, FALSE);
11    print_x_or_y (TRUE, FALSE);
12    print_x_or_y (FALSE, TRUE);
13    print_x_or_y (TRUE, TRUE);
14
15    print_x_or_y (TRUE, NULL);
16    print_x_or_y (FALSE, NULL);
17    print_x_or_y (NULL, TRUE);
18    print_x_or_y (NULL, FALSE);
19  END;
20  /
```

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

PL/SQL procedure successfully completed.

SQL>
PL/SQL Expressions and Comparisons

PL/SQL Language Fundamentals

PL/SQL procedure successfully completed.

SQL>

Topics:

■ Order of Evaluation
■ Short-Circuit Evaluation
■ Comparison Operators

Order of Evaluation
As with all operators, the order of evaluation for logical operators is determined by the
operator precedence shown in Table 2–2, and can be changed by parentheses, as in
Example 2–37. (Example 2–37 invokes the print_boolean procedure created in
Example 2–34.)

Example 2–37 Changing Order of Evaluation of Logical Operators

PL/SQL procedure successfully completed.

SQL>
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 errors.

In Example 2–38, short-circuit evaluation prevents the expression in line 8 from causing an error.

Example 2–38  Short-Circuit Evaluation

```
SQL> DECLARE
2    on_hand  INTEGER := 0;
3    on_order INTEGER := 100;
4  BEGIN
5    -- Does not cause divide-by-zero error;
6    -- evaluation stops after first expression
7    IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
8      DBMS_OUTPUT.PUT_LINE('On hand quantity is zero.');
9    END IF;
10   END;
11  /
On hand quantity is zero.
```

PL/SQL procedure successfully completed.

```sql
SQL>
```

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 either `TRUE`, `FALSE`, or `NULL`. Typically, you use comparison operators in conditional control statements and in the `WHERE` clauses of SQL data manipulation statements.

The comparison operators are:

- The relational operators summarized in Table 2–4
- `IS NULL Operator` on page 2-35
- `LIKE Operator` on page 2-35
- `BETWEEN Operator` on page 2-37
- `IN Operator` on page 2-37

Note: Using `CLOB` values with comparison operators can create temporary `LOB` values. Be sure that your temporary tablespace is large enough to handle them.
Example 2–39 invokes the `print_boolean` procedure created in Example 2–34 to print values of some expressions that include relational operators.

**Example 2–39  Relational Operators**

```sql
SQL> BEGIN
2    print_boolean ('(2 + 2 = 4)', 2 + 2 = 4);
3
4    print_boolean ('(2 + 2 <> 4)', 2 + 2 <> 4);
5    print_boolean ('(2 + 2 != 4)', 2 + 2 != 4);
6    print_boolean ('(2 + 2 ~= 4)', 2 + 2 ~= 4);
7    print_boolean ('(2 + 2 ^= 4)', 2 + 2 ^= 4);
8
9    print_boolean ('(1 < 2)', 1 < 2);
10
11   print_boolean ('(1 > 2)', 1 > 2);
12
13   print_boolean ('(1 <= 2)', 1 <= 2);
14
15   print_boolean ('(1 >= 1)', 1 >= 1);
16   END;
17
(2 + 2 = 4) = TRUE
(2 + 2 <> 4) = FALSE
(2 + 2 != 4) = FALSE
(2 + 2 ~= 4) = FALSE
(2 + 2 ^= 4) = FALSE
(1 < 2) = TRUE
(1 > 2) = FALSE
(1 <= 2) = TRUE
(1 >= 1) = TRUE

PL/SQL procedure successfully completed.

SQL>
```

**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 **NULL** values always yield **NULL**.

To test whether a value is **NULL**, use **IF value IS NULL**, as the procedure `print_boolean` in Example 2–34 does at line 6.

**LIKE Operator** The **LIKE** operator compares a character, string, or **CLOB** value to a pattern and returns **TRUE** if the value matches the pattern and **FALSE** if it does not.
The pattern can include the two "wildcard" characters underscore (_), and percent sign (%). Underscore matches exactly one character. Percent sign (%) matches zero or more characters.

Case is significant. The string 'Johnson' matches the pattern 'J%s_n' but not 'J%S_N', as Example 2–40 shows.

**Example 2–40  LIKE Operator**

```sql
SQL> DECLARE
2  PROCEDURE compare (value VARCHAR2, pattern VARCHAR2) IS
3    begin
4      if value LIKE pattern then
5        dbms_output.put_line ('true');
6      else
7        dbms_output.put_line ('false');
8      end if;
9    end;
10    begin
11      compare('Johnson', 'J%s_n');
12      compare('Johnson', 'J%S_N');
13    end;
14 /
true
false
```

PL/SQL procedure successfully completed.

SQL>

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

Example 2–41 uses the backslash as the escape character, so that the percent sign in the string does not act as a wildcard.

**Example 2–41  Escape Character in Pattern**

```sql
SQL> DECLARE
2  PROCEDURE half_off (sale_sign VARCHAR2) IS
3  begin
4    if sale_sign LIKE '50\% off!' ESCAPE '\' then
5      DBMS_OUTPUT.PUT_LINE ('true');
6    else
7      DBMS_OUTPUT.PUT_LINE ('false');
8    end if;
9  end;
10  begin
11    half_off('Going out of business!');
12    half_off('50% off!');
13  end;
14 /
false
true
```

2-36  Oracle Database PL/SQL Language Reference
PL/SQL procedure successfully completed.

SQL>

**BETWEEN Operator**  The BETWEEN operator tests whether a value lies in a specified range. \( x \) BETWEEN \( a \) AND \( b \) means that \( x \geq a \) and \( x \leq b \).

**Example 2–42** invokes the print_boolean procedure created in Example 2–34 to print values of some expressions that include the BETWEEN operator.

**Example 2–42  BETWEEN Operator**

```
SQL> BEGIN
  2    print_boolean ('2 BETWEEN 1 AND 3', 2 BETWEEN 1 AND 3);
  3    print_boolean ('2 BETWEEN 2 AND 3', 2 BETWEEN 2 AND 3);
  4    print_boolean ('2 BETWEEN 1 AND 2', 2 BETWEEN 1 AND 2);
  5    print_boolean ('2 BETWEEN 3 AND 4', 2 BETWEEN 3 AND 4);
  6  END;
  7  /
  8  2 BETWEEN 1 AND 3 = TRUE
  9  2 BETWEEN 2 AND 3 = TRUE
 10  2 BETWEEN 1 AND 2 = TRUE
 11  2 BETWEEN 3 AND 4 = FALSE
```

PL/SQL procedure successfully completed.

SQL>

**IN Operator**  The IN operator tests set membership. \( x \) IN (set) means that \( x \) is equal to any member of set.

**Example 2–43** invokes the print_boolean procedure created in Example 2–34 to print values of some expressions that include the IN operator.

**Example 2–43  IN Operator**

```
SQL> DECLARE
  2    letter VARCHAR2(1) := 'm';
  3  BEGIN
  4    print_boolean ('letter IN (''a'', ''b'', ''c'')', letter IN ('a', 'b', 'c'));
  5    letter IN (''z'', ''m'', ''y'', ''p'')', letter IN ('z', 'm', 'y', 'p'));
  6  END;
  7  /
  8  letter IN ('a', 'b', 'c') = FALSE
  9  letter IN ('z', 'm', 'y', 'p') = TRUE
```

PL/SQL procedure successfully completed.

SQL>

**Example 2–44** shows what happens when set contains a NULL value. (Example 2–44 invokes the print_boolean procedure created in Example 2–34.)
Example 2–44  Using the IN Operator with Sets with NULL Values

SQL> DECLARE
2    a INTEGER; -- Initialized to NULL by default
3    b INTEGER := 10;
4    c INTEGER := 100;
5  BEGIN
6    print_boolean ('100 IN (a, b, c)', 100 IN (a, b, c));
7    print_boolean ('100 NOT IN (a, b, c)', 100 NOT IN (a, b, c));
8
9    print_boolean ('100 IN (a, b)', 100 IN (a, b));
10   print_boolean ('100 NOT IN (a, b)', 100 NOT IN (a, b));
11
12   print_boolean ('a IN (a, b)', a IN (a, b));
13   print_boolean ('a NOT IN (a, b)', a NOT IN (a, b));
14  END;
15 /

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

PL/SQL procedure successfully completed.

SQL>

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.

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
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:

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

**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:

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

The following expression is true:

```plsql
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 Data Types 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 Data Type Conversion on page 3-28.

**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:

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

The following expression is true:
date1 > date2

Guidelines forBOOLEAN Expressions
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:

\[100 < \text{tax} < 500\]  -- not allowed

The debugged version follows:

\((100 < \text{tax}) \text{ AND } (\text{tax} < 500)\)

You can use a BOOLEAN variable itself as a condition; you need not compare it to the value TRUE or FALSE. In Example 2–45, the loops are equivalent.

**Example 2–45 Using BOOLEAN Variables in Conditional Tests**

```sql
SQL> DECLARE
2    done BOOLEAN;
3  BEGIN
4    -- The following WHILE loops are equivalent
5
6    done := FALSE;
7    WHILE done = FALSE
8      LOOP
9      done := TRUE;
10     END LOOP;
11
12    done := FALSE;
13    WHILE NOT (done = TRUE)
14      LOOP
15      done := TRUE;
16     END LOOP;
17
18    done := FALSE;
19    WHILE NOT done
20      LOOP
21      done := TRUE;
22     END LOOP;
23  END;
24 /

PL/SQL procedure successfully completed.
```

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 4-5.

Topics:
- Simple 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 2–46. 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 2–46   Using the WHEN Clause with a CASE Statement

```sql
SQL> DECLARE
2    grade CHAR(1) := 'B';
3    appraisal VARCHAR2(20);
4  BEGIN
5    appraisal :=
6      CASE grade
7        WHEN 'A' THEN 'Excellent'
8        WHEN 'B' THEN 'Very Good'
9        WHEN 'C' THEN 'Good'
10        WHEN 'D' THEN 'Fair'
11        WHEN 'F' THEN 'Poor'
12        ELSE 'No such grade'
13      END;
14    DBMS_OUTPUT.PUT_LINE
15      ('Grade ' || grade || ' is ' || appraisal);
16  END;
17  /
Grade B is Very Good
PL/SQL procedure successfully completed.
```

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 2–47.

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 2–47   Using a Search Condition with a CASE Statement

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

FUNCTION attends_this_school (id NUMBER)
RETURN BOOLEAN IS
BEGIN
  RETURN TRUE;
END;

BEGIN
  appraisal :=
  CASE
    WHEN attends_this_school(id) = FALSE
      THEN 'Student not enrolled'
    WHEN grade = 'F' OR attendance < min_days
      THEN 'Poor (poor performance or bad attendance)'
    WHEN grade = 'A' THEN 'Excellent'
    WHEN grade = 'B' THEN 'Very Good'
    WHEN grade = 'C' THEN 'Good'
    WHEN grade = 'D' THEN 'Fair'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE('Result for student ' || id || ' is ' || appraisal);
END;
/
Result for student 8429862 is Poor (poor performance or bad attendance)
PL/SQL procedure successfully completed.

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 using NULL values, remember the following rules:

- Comparisons involving NULL values always yield NULL.
- Applying the logical operator NOT to a NULL value 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. Instead, use a searched CASE syntax with WHEN expression IS NULL.

In Example 2–48, you might expect the sequence of statements to execute because \( x \) and \( y \) seem unequal. But, NULL values 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.
Example 2–48 NULL Value in Unequal Comparison

PL/SQL procedure successfully completed.

In Example 2–49, 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.

Example 2–49 NULL Value in Equal Comparison

PL/SQL procedure successfully completed.

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

NULL Values and the NOT Operator

Applying the logical operator NOT to a null yields NULL. Therefore, the following two IF statements are not always equivalent:

PL/SQL Expressions and Comparisons
2    x    INTEGER := 2;
3    Y    INTEGER := 5;
4    high INTEGER;
5  BEGIN
6     IF x > y THEN high := x;
7     ELSE high := y;
8     END IF;
9
10    IF NOT x > y THEN high := y;
11    ELSE high := x;
12    END IF;
13  END;
14  /

PL/SQL procedure successfully completed.

SQL>

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 value. This includes values returned
by character functions and BOOLEAN expressions. For example, the following
statements assign nulls to the target variables:

SQL> DECLARE
2     null_string  VARCHAR2(80) := TO_CHAR('');
3     address      VARCHAR2(80);
4     zip_code     VARCHAR2(80) := SUBSTR(address, 25, 0);
5     name         VARCHAR2(80);
6     valid        BOOLEAN      := (name != '');
7  BEGIN
8    NULL;
9  END;
10  /

PL/SQL procedure successfully completed.

SQL>

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:

SQL> BEGIN
2    DBMS_OUTPUT.PUT_LINE ('apple' || NULL || NULL || 'sauce');
3  END;
4  /

applesauce

PL/SQL procedure successfully completed.

SQL>
NULL Values as Arguments to Built-In Functions

If a NULL argument is passed to a built-in function, a NULL value 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 2–50, if the column `manager_id` is NULL, `DECODE` returns the value ‘nobody’.

**Example 2–50  NULL Value as Argument to DECODE Function**

```sql
SQL> DECLARE
2    manager  VARCHAR2(40);
3    name     employees.last_name%TYPE;
4  BEGIN
5    -- NULL is a valid argument to DECODE.
6    -- In this case, manager_id is NULL
7    -- and the DECODE function returns 'nobody'.
8
9    SELECT DECODE(manager_id, NULL, 'nobody', 'somebody'), last_name
10      INTO manager, name
11        FROM employees
12          WHERE employee_id = 100;
13
14    DBMS_OUTPUT.PUT_LINE
15      (name || ' is managed by ' || manager);
16  END;
17  /
King is managed by nobody
PL/SQL procedure successfully completed.
```

The function `NVL` returns the value of its second argument if its first argument is NULL. In Example 2–51, if the column specified in the query is NULL, the function returns the value -1 to signify a nonexistent employee in the output.

**Example 2–51  NULL Value as Argument to NVL Function**

```sql
SQL> DECLARE
2    manager employees.manager_id%TYPE;
3    name    employees.last_name%TYPE;
4  BEGIN
5    -- NULL is a valid argument to NVL.
6    -- In this case, manager_id is null
7    -- and the NVL function returns -1.
8
9    SELECT NVL(manager_id, -1), last_name
10      INTO manager, name
11        FROM employees
12          WHERE employee_id = 100;
13
14     DBMS_OUTPUT.PUT_LINE
15       (name || ' is managed by employee Id: ' || manager);
16  END;
17  /
King is managed by employee Id: -1
PL/SQL procedure successfully completed.
```
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 2–52 does not make any change to the value of `old_string`.

**Example 2–52  NULL Value as Second Argument to REPLACE Function**

```sql
SQL> DECLARE
2    string_type  VARCHAR2(60);
3    old_string   string_type%TYPE := 'Apples and oranges';
4    v_string     string_type%TYPE := 'more apples';
5
6    -- NULL is a valid argument to REPLACE,
7    -- but does not match anything,
8    -- so no replacement is done.
9
10   new_string string_type%TYPE := REPLACE(old_string, NULL, v_string);
11  BEGIN
12    DBMS_OUTPUT.PUT_LINE('Old string = ' || old_string);
13    DBMS_OUTPUT.PUT_LINE('New string = ' || new_string);
14  END;
15  /
Old string = Apples and oranges
New string = Apples and oranges
PL/SQL procedure successfully completed.
```

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

**Example 2–53  NULL Value as Third Argument to REPLACE Function**

```sql
SQL> DECLARE
2    string_type  VARCHAR2(60);
3    dashed       string_type%TYPE := 'Gold-i-locks';
4
5    -- When the substitution text for REPLACE is NULL,
6    -- the text being replaced is deleted.
7
8    name         string_type%TYPE := REPLACE(dashed, '-', NULL);
9  BEGIN
10     DBMS_OUTPUT.PUT_LINE('Dashed name    = ' || dashed);
11     DBMS_OUTPUT.PUT_LINE('Dashes removed = ' || name);
12  END;
13  /
Dashed name    = Gold-i-locks
Dashes removed = Goldilocks
PL/SQL procedure successfully completed.
```

If its third argument is `NULL`, `REPLACE` returns its first argument with every occurrence of its second argument removed. For example, the call to `REPLACE` in Example 2–53 removes all the dashes from `dashed_string`, instead of changing them to another character.
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 13-125 and SQLERRM Function on page 13-126.

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
  - RAWT_HEX
  - 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 RAWT_HEX function accepts an argument of data type 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

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 \texttt{PLSQL_CCFLAGS} compilation parameter, which is described in PL/SQL Units and Compilation Parameters on page 1-25. Selection directives can test inquiry directives or static package constants.

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

\textbf{Note:} The conditional compilation feature and related PL/SQL packages are available for Oracle Database 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
- Using DBMS_DB_VERSION Package Constants

Conditional Compilation Control Tokens

The conditional compilation trigger character, $, identifies code that is processed before the application is compiled. A conditional compilation control token has the form:

\[
\text{preprocessor_control_token ::= $plsql_identifier}
\]
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 2–56 on page 2-54.

**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 2-50. For information about 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 exception 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 2-50. See Example 2–55.

**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 2-50 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 $$id for the search key.
2. The two-pass algorithm proceeds as follows:
   - The string in the PLSQL_CCFLAGS compilation parameter is scanned from right to left, searching with id for a matching name (case-insensitive); done if found.
   - The predefined inquiry directives are searched; done if found.
3. If the $$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. If the PL/SQL code is wrapped, then the warning message is disabled so that the undefined inquiry directive is not revealed.

For example, consider 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. 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.

Consider the following session setting:
ALTER SESSION SET PLSQL_CCFLAGS = 'debug:true'

Now the value of $$debug is true, the value of $$plsql_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 2–56 on page 2-54.

Using Predefined Inquiry Directives with Conditional Compilation
Predefined inquiry directive names, which can be used in conditional expressions, include:

- **PLSQL_LINE**, a PLS_INTEGER literal whose value indicates the line number reference to $$PLSQL_LINE in the current PL/SQL unit
  
  An example of $$PLSQL_LINE in a conditional expression is:
  
  $IF $$PLSQL_LINE = 32 $THEN ... 

- **PLSQL_UNIT**, a VARCHAR2 literal whose value indicates the current PL/SQL unit
  
  For a named PL/SQL unit, $$PLSQL_UNIT contains, but might not be limited to, the unit name. For an anonymous block, $$PLSQL_UNIT contains the empty string.
  
  An example of $$PLSQL_UNIT in a conditional expression is:
  
  IF $$PLSQL_UNIT = 'AWARD_BONUS' THEN ... 

  The preceding 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, as follows:
  
  $IF $$PLSQL_UNIT IS NULL $THEN ...

- **PL/SQL compilation parameters**

  The values of the literals PLSQL_LINE and PLSQL_UNIT can be defined explicitly with the compilation parameter PLSQL_CCFLAGS. For information about compilation parameters, see PL/SQL Units and Compilation Parameters on page 1-25.

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 about PL/SQL data types, see Predefined PL/SQL Scalar Data Types 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:
    - \(x > y\)
    - \(x < y\)
    - \(x \geq y\)
    - \(x \leq y\)
    - \(x = y\)
    - \(x \neq y\)
  ■ Where \(x\) and \(y\) are PLS_INTEGER BOOLEAN expressions:
    - \(\text{NOT } x\)
    - \(x \text{ AND } y\)
    - \(x \text{ OR } y\)
    - \(x > y\)
    - \(x \geq y\)
    - \(x = y\)
    - \(x \leq y\)
    - \(x \neq y\)
  ■ Where \(x\) is a static expression:
    - \(x \text{ IS NULL}\)
    - \(x \text{ IS NOT NULL}\)

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
  ■ \(x \text{ || } y\) 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 data_type := static_expression;
```

This is a valid declaration of a static constant if:

- The declared `data_type` and the type of `static_expression` are the same
- `static_expression` is a static expression
- `data_type` 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 must be recompiled to pick up any changes. 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. For control of conditional compilation in an individual unit, you can set a specific flag in the PL/SQL compilation parameter `PLSQL_CCFLAGS`. For information about PL/SQL compilation parameters, see PL/SQL Units and Compilation Parameters on page 1-25.

In Example 2–54 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 2–54 Using Static Constants**

```
SQL> CREATE PACKAGE my_debug IS
    2    debug CONSTANT BOOLEAN := TRUE;
    3    trace CONSTANT BOOLEAN := TRUE;
    4  END my_debug;
    5  /

Package created.

SQL> CREATE PROCEDURE my_proc1 IS
    2  BEGIN
    3    $IF my_debug.debug $THEN
    4      DBMS_OUTPUT.put_line('Debugging ON');
    5    $ELSE
    6      DBMS_OUTPUT.put_line('Debugging OFF');
    7    $END
    8  END my_proc1;
    9  /

Procedure created.

SQL> CREATE PROCEDURE my_proc2 IS
    2  BEGIN
    3    $IF my_debug.trace $THEN
    4      DBMS_OUTPUT.put_line('Tracing ON');
    5    $ELSE DBMS_OUTPUT.put_line('Tracing OFF');
    6    $END
```
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.

### 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 Database 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 Database 10g release 1 or earlier are `FALSE`.

**Example 2–55** 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 2–55  Using DBMS_DB_VERSION Constants

```sql
SQL> BEGIN
2    $IF DBMS_DB_VERSION.VER_LE_10_1 $THEN
3      $ERROR 'unsupported database release'
4    $END
5    $ELSE
6      DBMS_OUTPUT.PUT_LINE
7        ('Release ' || DBMS_DB_VERSION.VERSION || '.' ||
8          DBMS_DB_VERSION.RELEASE || ' is supported. ');
9
10    -- This COMMIT syntax is newly supported in 10.2:
11    COMMIT WRITE IMMEDIATE NOWAIT;
12    $END
13  $END;
14  /
Release 11.1 is supported.
PL/SQL procedure successfully completed.
SQL>
```

For information about 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

Example 2–56 uses conditional compilation to determine whether the \texttt{BINARY\_DOUBLE} data type can be used in the calculations for PL/SQL units in the database. The \texttt{BINARY\_DOUBLE} data type can only be used in a database version that is 10g or later.

\textbf{Example 2–56 Using Conditional Compilation with Database Versions}

\begin{verbatim}
SQL> -- Set flags for displaying debugging code and tracing info:
SQL>
SQL> ALTER SESSION SET PLSQL_CCFLAGS =
2    'my_debug:FALSE, my_tracing:FALSE';
Session altered.

SQL> CREATE OR REPLACE PACKAGE my_pkg AS
2    SUBTYPE my_real IS
3      $IF DBMS_DB_VERSION.VERSION < 10 $THEN
4        NUMBER;
5        -- Check database version
6      $ELSE
7        BINARY_DOUBLE;
8      $END
9
10      my_pi my_real;
11      my_e my_real;
12  END my_pkg;
13  /
Package created.

SQL> CREATE OR REPLACE PACKAGE BODY my_pkg AS
2  BEGIN
3    -- Set values for future calculations based on DB version
4
5    $IF DBMS_DB_VERSION.VERSION < 10 $THEN
6      my_pi := 3.14159265358979323846264338327950288420;
7      my_e  := 2.71828182845904523536028747135266249775;
8    $ELSE
9      my_pi := 3.14159265358979323846264338327950288420d;
10     my_e  := 2.71828182845904523536028747135266249775d;
11    $END
12  END my_pkg;
13  /
Package body created.

SQL> CREATE OR REPLACE PROCEDURE circle_area(radius my_pkg.my_real) IS
2    my_area      my_pkg.my_real;
3    my_data_type  VARCHAR2(30);
\end{verbatim}
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 debugging code
  SELECT DATA_TYPE INTO my_data_type
    FROM USER_ARGUMENTS
    WHERE OBJECT_NAME = 'CIRCLE_AREA'
    AND ARGUMENT_NAME = 'RADIUS';
  DBMS_OUTPUT.PUT_LINE ('Data type of the RADIUS argument is: ' || my_data_type);
$END
END;
/

Procedure created.

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 2–57 shows how to print the post-processed form of my_pkg in Example 2–56 with the PRINT_POST_PROCESSED_SOURCE procedure.

Example 2–57 Using PRINT_POST_PROCESSED_SOURCE to Display Source Code

CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE ('PACKAGE', 'HR', 'MY_PKG');
PACKAGE my_pkg AS
  SUBTYPE my_real IS BINARY_DOUBLE;
  my_pi my_real;
  my_e my_real;
END my_pkg;

Call completed.

PRINT_POST_PROCESSED_SOURCE replaces unselected text with whitespace. The lines of code in Example 2–56 that are not included in the post-processed text are represented as blank lines. For information about 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:

```sql
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:

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

## Using PL/SQL to Create Web Applications

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

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

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

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

See Also: Oracle Database Advanced Application Developer’s Guide for information about creating web applications

Using PL/SQL to Create 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.

See Also: Oracle Database Advanced Application Developer’s Guide for information about creating web server pages
Every constant, variable, and parameter has a **data type** (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 data types and subtypes, and lets you define your own PL/SQL subtypes.

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

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

**Table 3–1** lists the categories of predefined PL/SQL data types, describes the data they store, and tells where to find information about the specialized data types.

**Table 3–1  Categories of Predefined PL/SQL Data Types**

<table>
<thead>
<tr>
<th>Data Type Category</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>Single values with no internal components.</td>
</tr>
<tr>
<td>Composite</td>
<td>Data items that have internal components that can be accessed individually. Explained in Chapter 5, &quot;Using PL/SQL Collections and Records.&quot;</td>
</tr>
<tr>
<td>Reference</td>
<td>Pointers to other data items. Explained in <strong>Using Cursor Variables (REF CURSORs)</strong> on page 6-22.</td>
</tr>
<tr>
<td>Large Object (LOB)</td>
<td>Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.</td>
</tr>
</tbody>
</table>

**Topics:**
- Predefined PL/SQL Scalar Data Types and Subtypes
- Predefined PL/SQL Large Object (LOB) Data Types
- User-Defined PL/SQL Subtypes
- PL/SQL Data Type Conversion

**Predefined PL/SQL Scalar Data Types and Subtypes**

Scalar data types store single values with no internal components. **Table 3–2** lists the predefined PL/SQL scalar data types and describes the data they store.
Predefined PL/SQL Scalar Data Types and Subtypes

Topics:
- Predefined PL/SQL Numeric Data Types and Subtypes
- Predefined PL/SQL Character Data Types and Subtypes
- Predefined PL/SQL BOOLEAN Data Type
- Predefined PL/SQL Datetime and Interval Data Types

Predefined PL/SQL Numeric Data Types and Subtypes

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLS_INTEGER or BINARY_INTEGER</td>
<td>Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits</td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td>Single-precision IEEE 754-format floating-point number</td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td>Double-precision IEEE 754-format floating-point number</td>
</tr>
<tr>
<td>NUMBER</td>
<td>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.</td>
</tr>
</tbody>
</table>

Topics:
- PLS_INTEGER and BINARY_INTEGER Data Types
- SIMPLE_INTEGER Subtype of PLS_INTEGER
- BINARY_FLOAT and BINARY_DOUBLE Data Types
- NUMBER Data Type

PLS_INTEGER and BINARY_INTEGER Data Types

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

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

The PLS_INTEGER data type has the following advantages over the NUMBER data type and NUMBER subtypes:
- PLS_INTEGER values require less storage.
PLS_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic.

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

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

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

### Table 3–4 Predefined Subtypes of PLS_INTEGER Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATURAL</td>
<td>Nonnegative PLS_INTEGER value</td>
</tr>
<tr>
<td>NATURALN</td>
<td>Nonnegative PLS_INTEGER value with NOT NULL constraint</td>
</tr>
<tr>
<td>POSITIVE</td>
<td>Positive PLS_INTEGER value</td>
</tr>
<tr>
<td>POSITIVEN</td>
<td>Positive PLS_INTEGER value with NOT NULL constraint</td>
</tr>
<tr>
<td>SIGNTYPE</td>
<td>PLS_INTEGER value -1, 0, or 1 (useful for programming tri-state logic)</td>
</tr>
<tr>
<td>SIMPLE_INTEGER</td>
<td>PLS_INTEGER value with NOT NULL constraint</td>
</tr>
</tbody>
</table>

**SIMPLE_INTEGER** Subtype of PLS_INTEGER

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

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

Topics:

- Overflow Semantics
- Overloading Rules
- Integer Literals
- Cast Operations
- Compiler Warnings

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

```sql
SQL> DECLARE
     2   n SIMPLE_INTEGER := 2147483645;
```
Predefined PL/SQL Scalar Data Types and Subtypes

```sql
3  BEGIN
4    FOR j IN 1..4 LOOP
5      n := n + 1;
6      DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
7    END LOOP;
8    FOR j IN 1..4 LOOP
9      n := n - 1;
10     DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
11    END LOOP;
12  END;
13  /
14
+2147483646
+2147483647
-2147483648
-2147483647
-2147483648
+2147483647
+2147483646
+2147483645

PL/SQL procedure successfully completed.
```

SQL>

Overloading Rules

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

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

**Integer Literals** Integer literals in the `SIMPLE_INTEGER` range have the 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` data type makes no conversion if the source value is not `NULL`. If the source value is `NULL`, a run-time exception is raised.

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

**Compiler Warnings**
The compiler issues a warning in 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 Data Types**
The `BINARY_FLOAT` and `BINARY_DOUBLE` data types represent single-precision and double-precision IEEE 754-format floating-point numbers, respectively.

A `BINARY_FLOAT` literal ends with `f` (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:

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

**Table 3–5  Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY_FLOAT_NAN</td>
<td><code>BINARY_FLOAT</code> value for which the condition IS NAN (not a number) is true</td>
</tr>
<tr>
<td>BINARY_FLOAT_INFINITY</td>
<td>Single-precision positive infinity</td>
</tr>
<tr>
<td>BINARY_FLOAT_MAX_NORMAL</td>
<td>Maximum normal <code>BINARY_FLOAT</code> value</td>
</tr>
<tr>
<td>BINARY_FLOAT_MIN_NORMAL</td>
<td>Minimum normal <code>BINARY_FLOAT</code> value</td>
</tr>
<tr>
<td>BINARY_FLOAT_MAX_SUBNORMAL</td>
<td>Maximum subnormal <code>BINARY_FLOAT</code> value</td>
</tr>
<tr>
<td>BINARY_FLOAT_MIN_SUBNORMAL</td>
<td>Minimum subnormal <code>BINARY_FLOAT</code> value</td>
</tr>
<tr>
<td>BINARY_DOUBLE_NAN</td>
<td><code>BINARY_DOUBLE</code> value for which the condition IS NAN (not a number) is true</td>
</tr>
<tr>
<td>BINARY_DOUBLE_INFINITY</td>
<td>Double-precision positive infinity</td>
</tr>
</tbody>
</table>
In the IEEE-754 standard, subnormal ranges of values are intended to reduce problems caused by underflow to zero.

BINARY_FLOAT and BINARY_DOUBLE data types are primarily for high-speed scientific computation, as explained in Writing Computation-Intensive PL/SQL Programs on page 12-27.

See Also: Guidelines for Overloading with Numeric Types on page 8-13, for information about writing libraries that accept different numeric types.

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

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

**NUMBER Data Type**

The NUMBER data type stores fixed-point or floating-point numbers with absolute values in the range \(1 \times 10^{-130}\) up to (but not including) \(1.0 \times 10^{126}\). 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:

\[ \text{NUMBER}(\text{precision}, \text{scale}) \]

For example:

\[ \text{NUMBER}(8,2) \]

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

\[ \text{NUMBER}(\text{precision}) \]
NUMBER(precision)

For example:

NUMBER(2)

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

NUMBER

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

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

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

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

For more information about the NUMBER data type, see Oracle Database SQL Language Reference.

Table 3–6 lists the predefined subtypes of the NUMBER data type and describes the data they store.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC, DECIMAL, or NUMERIC</td>
<td>Fixed-point NUMBER with maximum precision of 38 decimal digits</td>
</tr>
<tr>
<td>DOUBLE PRECISION or FLOAT</td>
<td>Floating-point NUMBER with maximum precision of 126 binary digits (approximately 38 decimal digits)</td>
</tr>
<tr>
<td>INT, INTEGER, or SMALLINT</td>
<td>Integer with maximum precision of 38 decimal digits</td>
</tr>
<tr>
<td>REAL</td>
<td>Floating-point NUMBER with maximum precision of 63 binary digits (approximately 18 decimal digits)</td>
</tr>
</tbody>
</table>

Predefined PL/SQL Character Data Types and Subtypes

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>Fixed-length character string with maximum size of 32,767 bytes</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>Variable-length character string with maximum size of 32,767 bytes</td>
</tr>
<tr>
<td>RAW</td>
<td>Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL</td>
</tr>
<tr>
<td>NCHAR</td>
<td>Fixed-length national character string with maximum size of 32,767 bytes</td>
</tr>
</tbody>
</table>
CHAR and VARCHAR2 Data Types

The CHAR and VARCHAR2 data types store fixed-length and variable-length character strings, respectively. All string literals have data type CHAR.

How CHAR and VARCHAR2 data is represented internally depends on the database character set 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:

\[ [ \text{CHAR} | \text{VARCHAR2} ] [ ( \text{maximum_size} [ \text{CHAR} | \text{BYTE} ] ) ] \]

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:

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

the following assignment raises VALUE_ERROR:

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

If the character value that you insert into a database column is longer than the defined width of the column, PL/SQL does not truncate the value or strip trailing blanks; it stops the insertion and raises an exception. To strip trailing blanks from a character value before assigning it to a variable or inserting it into a database column, use the built-in function RTRIM. For example, given the preceding declaration, the following assignment does not raise an exception:

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

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

**Differences Between CHAR and VARCHAR2 Data Types**

CHAR and VARCHAR2 data types differ in the following:

- Predefined Subtypes of Character Data Types
- Memory Allocation for Character Variables
- Blank-Padding Shorter Character Values
- Comparing Character Values
- Maximum Sizes of Values Inserted into Character Database Columns

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

---

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

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

For example, if you assign the same 500-byte value to VARCHAR2(1999 BYTE) and VARCHAR2(2000 BYTE) variables, PL/SQL allocates 1999 bytes for the former variable at compile time and 500 bytes for the latter variable at run time.
Blank-Padding Shorter Character Values  In each of the following situations, whether or not PL/SQL blank-pads the character value depends on the data type of the receiver:

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

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

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

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

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

Comparing Character Values  You can use relational operators in Table 2–4 on page 2-35 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:

```sql
SQL> DECLARE
2  last_name1 VARCHAR2(10) := 'COLES';
3  last_name2 VARCHAR2(10) := 'COLEMAN';
4  BEGIN
5    IF last_name1 > last_name2
6      THEN
7        DBMS_OUTPUT.PUT_LINE
8          (last_name1 || ' is greater than ' || last_name2);
9      ELSE
10        DBMS_OUTPUT.PUT_LINE
11          (last_name2 || ' is greater than ' || last_name1);
12    END IF;
13  END;
14  /
COLES is greater than COLEMAN
```

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

If both values have data type **CHAR**, PL/SQL blank-pads the shorter value to the length of the longer value before comparing them. In **Example 3–1**, the IF condition is TRUE.

If either value has data type **VARCHAR2**, PL/SQL does not adjust their lengths before comparing them. In both **Example 3–2** and **Example 3–3**, the IF condition is FALSE.

**Example 3–1  Comparing Two CHAR Values**

```sql
SQL> DECLARE
2  last_name1 CHAR(5)  := 'BELLO';     -- no trailing blanks
3  last_name2 CHAR(10) := 'BELLO   ';  -- trailing blanks
4  BEGIN
5  
```
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;
/
BELLO is equal to BELLO

PL/SQL procedure successfully completed.

Example 3–2 Comparing Two VARCHAR2 Values

DECLARE
  last_name1 VARCHAR2(10) := 'DOW';     -- no trailing blanks
  last_name2 VARCHAR2(10) := 'DOW   ';  -- trailing blanks
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
    (last_name1 || ' is equal to ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
    (last_name2 || ' is not equal to ' || last_name1);
  END IF;
END;
/
DOW    is not equal to DOW

PL/SQL procedure successfully completed.

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;
/
STAUB      is not equal to STAUB

PL/SQL procedure successfully completed.

Maximum Sizes of Values Inserted into Character Database Columns

The largest CHAR value that you can insert into a CHAR database column is 2,000 bytes.
The largest **VARCHAR2** value that you can insert into a **VARCHAR2** database column is 4,000 bytes.

You can insert any **CHAR** or **VARCHAR2** value into a **LONG** database column, because the maximum width of a **LONG** column is 2,147,483,648 bytes (2 GB). However, you cannot retrieve a value longer than 32,767 bytes from a **LONG** column into a **CHAR** or **VARCHAR2** variable. (The **LONG** data type is supported only for backward compatibility with existing applications. For more information, see **LONG** and **LONG RAW** Data Types on page 3-14.)

**RAW** Data Type

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

The syntax for specifying a **RAW** data item is:

```
RAW (maximum_size)
```

For example:

```
RAW(256)
```

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

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

You can insert any **RAW** value into a **LONG RAW** database column, because the maximum width of a **LONG RAW** column is 2,147,483,648 bytes (2 GB). However, you cannot retrieve a value longer than 32,767 bytes from a **LONG RAW** column into a **RAW** variable. (The **LONG RAW** data type is supported only for backward compatibility with existing applications. For more information, see **LONG** and **LONG RAW** Data Types on page 3-14.)

**NCHAR** and **NVARCHAR2** Data Types

The **NCHAR** and **NVARCHAR2** data types store fixed-length and variable-length national character strings, respectively.

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** Data Type
- **NVARCHAR2** Data Type

**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.
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 Data Type

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

The syntax for specifying an NCHAR data item is:

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

For example:

```
NCHAR
NCHAR(100)
```

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

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

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

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

---

**Table 3–8 Comparison of AL16UTF16 and UTF8 Encodings**

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Character Size (Bytes)</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL16UTF16</td>
<td>2</td>
<td>Easy to calculate string lengths, which you must do in order to avoid truncation errors when mixing programming languages.</td>
<td>Strings composed mostly of ASCII or EBCDIC characters take more space than necessary.</td>
</tr>
<tr>
<td>UTF8</td>
<td>1, 2, or 3</td>
<td>If most characters use only one byte, you can fit more characters into a variable or table column.</td>
<td>Possibility of truncation errors when transferring the data to a buffer measured in bytes.</td>
</tr>
</tbody>
</table>
**NVARCHAR2 Data Type**  The NVARCHAR2 data type stores variable-length national character strings. Because this type can always accommodate multiple-byte characters, you can use it to store any Unicode character data.

The syntax for specifying an NVARCHAR2 data item is:

```sql
NVARCHAR2 (maximum_size)
```

For example:

```sql
NVARCHAR2(300)
```

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

The largest NVARCHAR2 value that you can insert into an NVARCHAR2 database column is 4,000 bytes.

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

**LONG and LONG RAW Data Types**

---

**Note:** The LONG and LONG RAW data types are supported only for backward compatibility with existing applications. For new applications, use CLOB or NCLOB instead of LONG, and BLOB or BFILE instead of LONG RAW. Oracle recommends that you also replace existing LONG and LONG RAW data types with LOB data types. See *Predefined PL/SQL Large Object (LOB) Data Types* on page 3-22.

---

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

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

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

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

**See Also:** Oracle Database SQL Language Reference for information about referencing LONG columns in SQL statements

**ROWID and UROWID Data Types**

Internally, every database table has a ROWID pseudocolumn, which 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
Predefined PL/SQL Scalar Data Types and Subtypes

index-organized table. The ROWID data type can store only physical rowids, while the UROWID (universal rowid) data type can store physical, logical, or foreign (not database) rowids.

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

Physical rowids are useful for fetching across commits, as in Example 6–42 on page 6-40.

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.

Predefined PL/SQL BOOLEAN Data Type

The BOOLEAN data type stores logical values, which you can use in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL. The syntax for specifying an BOOLEAN data item is:

BOOLEAN

SQL has no data type equivalent to BOOLEAN; therefore you cannot use BOOLEAN variables or parameters in the following:

- SQL statements
- Built-in SQL functions (such as TO_CHAR)
- PL/SQL functions invoked from SQL statements

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 Data Types

The data types in this section let you store and manipulate dates, times, and intervals (periods of time). A variable that has a date and time data type stores values called datetimes. A variable that has an interval data type 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:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Valid Datetime Values</th>
<th>Valid Interval Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>-4712 to 9999 (excluding year 0)</td>
<td>Any nonzero integer</td>
</tr>
<tr>
<td>MONTH</td>
<td>01 to 12</td>
<td>0 to 11</td>
</tr>
<tr>
<td>DAY</td>
<td>01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)</td>
<td>Any nonzero integer</td>
</tr>
<tr>
<td>HOUR</td>
<td>00 to 23</td>
<td>0 to 23</td>
</tr>
<tr>
<td>MINUTE</td>
<td>00 to 59</td>
<td>0 to 59</td>
</tr>
<tr>
<td>SECOND</td>
<td>00 to 59.9(n), where 9(n) is the precision of time fractional seconds</td>
<td>0 to 59.9(n), where 9(n) is the precision of interval fractional seconds</td>
</tr>
<tr>
<td>TIMEZONE_HOUR</td>
<td>-12 to 14 (range accommodates daylight savings time changes)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>TIMEZONE_MINUTE</td>
<td>00 to 59</td>
<td>Not applicable</td>
</tr>
<tr>
<td>TIMEZONE_REGION</td>
<td>Found in the dynamic performance view V$TIMEZONE_NAMES</td>
<td>Not applicable</td>
</tr>
<tr>
<td>TIMEZONE_ABBR</td>
<td>Found in the dynamic performance view V$TIMEZONE_NAMES</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

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 Data Type**
- **TIMESTAMP Data Type**
- **TIMESTAMP WITH TIME ZONE Data Type**
- **TIMESTAMP WITH LOCAL TIME ZONE Data Type**
- **INTERVAL YEAR TO MONTH Data Type**
- **INTERVAL DAY TO SECOND Data Type**
- **Datetime and Interval Arithmetic**
- **Avoiding Truncation Problems Using Date and Time Subtypes**

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

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

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

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

You can add and subtract dates. In arithmetic expressions, PL/SQL interprets integer literals as days. For example, `SYSDATE + 1` signifies the same time tomorrow.

**TIMESTAMP Data Type**

The data type `TIMESTAMP`, which extends the data type `DATE`, stores the year, month, day, hour, minute, and second. The syntax is:

```sql
TIMESTAMP[(precision)]
```

where the optional parameter `precision` specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

The default timestamp format is set by the Oracle initialization parameter `NLS_TIMESTAMP_FORMAT`.

**Example 3–4** declares a variable of type `TIMESTAMP` and assigns a literal value to it. The fractional part of the seconds field is 0.275.

```sql
Example 3–4 Assigning a Literal Value to a TIMESTAMP Variable

SQL> DECLARE
  2    checkout TIMESTAMP(3);
  3  BEGIN
  4    checkout := '22-JUN-2004 07:48:53.275';
  5    DBMS_OUTPUT.PUT_LINE( TO_CHAR(checkout));
  6  END;
  7  /
22-JUN-04 07.48.53.275 AM
PL/SQL procedure successfully completed.

SQL>
```

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**

```sql
Example 3–5 Using the SCN_TO_TIMESTAMP and TIMESTAMP_TO_SCN Functions

SQL> DECLARE
  2    right_now TIMESTAMP;
  3    yesterday TIMESTAMP;
  4    sometime TIMESTAMP;
  5    scn1 INTEGER;
  6    scn2 INTEGER;
```
TIMESTAMP WITH TIME ZONE Data Type

The data type `TIMESTAMP WITH TIME ZONE`, which extends the data type `TIMESTAMP`, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly Greenwich Mean Time (GMT). The syntax is:

```
TIMESTAMP[(precision)] WITH TIME ZONE
```

where the optional parameter `precision` specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

The default timestamp with time zone format is set by the Oracle initialization parameter `NLS_TIMESTAMP_TZ_FORMAT`.

Example 3–6 declares a variable of type `TIMESTAMP WITH TIME ZONE` and assign a literal value to it. The time-zone displacement is +02:00.

Example 3–6  Assigning a Literal to a `TIMESTAMP WITH TIME ZONE` Variable

```sql
SQL> DECLARE
2  logoff TIMESTAMP(3) WITH TIME ZONE;
3  BEGIN
4    logoff := '10-OCT-2004 09:42:37.114 AM +02:00';
5    DBMS_OUTPUT.PUT_LINE (TO_CHAR(logoff));
6  END;
7  /
10-OCT-04 09.42.37.114 AM +02:00
PL/SQL procedure successfully completed.
```

SQL>
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.

```sql
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` 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 Data Type**

The data type `TIMESTAMP WITH LOCAL TIME ZONE`, which extends the data type `TIMESTAMP`, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC)—formerly Greenwich Mean Time. You can also use named time zones, as with `TIMESTAMP`.

The syntax is:

```
TIMESTAMP[(precision)] WITH LOCAL TIME ZONE
```

where the optional parameter `precision` specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

This data type differs from `TIMESTAMP WITH TIME ZONE` in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, Oracle returns it in your local session time zone.

Both Example 3–7 and Example 3–8 declare a variable of type `TIMESTAMP WITH LOCAL TIME ZONE` and assign it a value. The value in Example 3–7 is an appropriate local time, but the value in Example 3–8 includes a time zone displacement, which causes an error.

**Example 3–7  Correct Assignment to TIMESTAMP WITH LOCAL TIME ZONE**

```sql
SQL> DECLARE
2   logoff  TIMESTAMP(3) WITH LOCAL TIME ZONE;
3  BEGIN
4    logoff := '10-OCT-2004 09:42:37.114 AM ';
5    DBMS_OUTPUT.PUT_LINE(TO_CHAR(logoff));
6  END;
7 / 10-OCT-04 09.42.37.114 AM
PL/SQL procedure successfully completed.
```

**Example 3–8  An Inappropriate Assignment to TIMESTAMP WITH LOCAL TIME ZONE**

```sql
SQL> DECLARE
2   logoff  TIMESTAMP (3) WITH LOCAL TIME ZONE;
3  BEGIN
4    logoff := '29-AUG-2004 08:00:00 -8:00';
5    DBMS_OUTPUT.PUT_LINE(TO_CHAR(logoff));
6  END;
7 /
```

Oracle returns an error.
Example 3–8  Incorrect Assignment to TIMESTAMP WITH LOCAL TIME ZONE

SQL> DECLARE
2    logoff  TIMESTAMP(3) WITH LOCAL TIME ZONE;
3  BEGIN
4    logoff := '10-OCT-2004 09:42:37.114 AM +02:00';
5  END;
6  /
DECLARE
*
ERROR at line 1:
ORA-01830: date format picture ends before converting entire input string
ORA-06512: at line 4

SQL>

INTERVAL YEAR TO MONTH Data Type

Use the data type 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.

Example 3–9 declares a variable of type INTERVAL YEAR TO MONTH and assigns a value of 101 years and 3 months to it, in three different ways.

Example 3–9  Assigning Literals to an INTERVAL YEAR TO MONTH Variable

SQL> DECLARE
2    lifetime  INTERVAL YEAR(3) TO MONTH;
3  BEGIN
4    lifetime := INTERVAL '101-3' YEAR TO MONTH;  -- Interval literal
5
6    lifetime := '101-3';  -- Implicit conversion from character type
7
8    lifetime := INTERVAL '101' YEAR;  -- Specify only years
9    lifetime := INTERVAL '3' MONTH;   -- Specify only months
10  END;
11  /

PL/SQL procedure successfully completed.

SQL>

INTERVAL DAY TO SECOND Data Type

You use the data type 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.
Example 3–10 declares a variable of type INTERVAL DAY TO SECOND and assigns a value to it.

**Example 3–10 Assigning Literals to an INTERVAL DAY TO SECOND Variable**

```sql
SQL> DECLARE
2    lag_time  INTERVAL DAY(3) TO SECOND(3);
3  BEGIN
4    lag_time := '7 09:24:30';
5
6    IF lag_time > INTERVAL '6' DAY THEN
7      DBMS_OUTPUT.PUT_LINE ('Greater than 6 days');
8    ELSE
9      DBMS_OUTPUT.PUT_LINE ('Less than 6 days');
10    END IF;
11  END;
12  /
Greater than 6 days

PL/SQL procedure successfully completed.

SQL>
```

**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:

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td>+</td>
<td>interval</td>
<td>datetime</td>
</tr>
<tr>
<td>datetime</td>
<td>-</td>
<td>interval</td>
<td>datetime</td>
</tr>
<tr>
<td>interval</td>
<td>+</td>
<td>datetime</td>
<td>datetime</td>
</tr>
<tr>
<td>datetime</td>
<td>-</td>
<td>datetime</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>+</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>-</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>*</td>
<td>numeric</td>
<td>interval</td>
</tr>
<tr>
<td>numeric</td>
<td>*</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>interval</td>
<td>/</td>
<td>numeric</td>
<td>interval</td>
</tr>
</tbody>
</table>

**See Also:** *Oracle Database SQL Language Reference* for information about using SQL functions to perform arithmetic operations on datetime values

**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
Predefined PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types reference large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Predefined PL/SQL LOB data types are listed and described in Table 3–9.

Table 3–9 Predefined PL/SQL Large Object (LOB) Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>Used to store large binary objects in operating system files outside the database.</td>
<td>System-dependent. Cannot exceed 4 gigabytes (GB).</td>
</tr>
<tr>
<td>BLOB</td>
<td>Used to store large binary objects in the database.</td>
<td>8 to 128 terabytes (TB)</td>
</tr>
<tr>
<td>CLOB</td>
<td>Used to store large blocks of character data in the database.</td>
<td>8 to 128 TB</td>
</tr>
<tr>
<td>NCLOB</td>
<td>Used to store large blocks of NCHAR data in the database.</td>
<td>8 to 128 TB</td>
</tr>
</tbody>
</table>

LOB Locators

To reference a large object that is stored in an external file, a LOB data type uses a LOB locator, which is stored in an external file, either inside the row (inline) or outside the row (out-of-line). In the external file, LOB locators are in columns of the types BFILE, BLOB, CLOB, and NCLOB.

PL/SQL operates on large objects through their LOB locators. For example, when you select a BLOB column value, PL/SQL returns only its locator. If PL/SQL returned the locator during a transaction, the locator includes a transaction ID, so you cannot use that locator to update that large object in another transaction. Likewise, you cannot save a locator during one session and then use it in another session.

Differences Between LOB Data Types and LONG and LONG RAW Data Types

LOB data types differ from LONG and LONG RAW data types in the following ways:

<table>
<thead>
<tr>
<th>Difference</th>
<th>LOB Data Types</th>
<th>LONG and LONG RAW Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>Functionality enhanced in every release.</td>
<td>Functionality static. Supported only for backward compatibility with existing applications.</td>
</tr>
<tr>
<td>Maximum size</td>
<td>8 to 128 TB</td>
<td>2 GB</td>
</tr>
<tr>
<td>Access</td>
<td>Random</td>
<td>Sequential</td>
</tr>
<tr>
<td>Can be object type attribute</td>
<td>BFILE, BLOB, CLOB: YesNCLOB: No</td>
<td>No</td>
</tr>
</tbody>
</table>

See Also:
- LONG and LONG RAW Data Types on page 3-14
- Oracle Database SecureFiles and Large Objects Developer’s Guide for more information about LOBs
Topics:
- BFILE Data Type
- BLOB Data Type
- CLOB Data Type
- NCLOB Data Type

BFILE Data Type
You use the BFILE data type to store large binary objects in operating system files outside the database. Every BFILE variable stores a file locator, which points to a large binary file on the server. The locator includes a directory alias, which specifies a full path name. Logical path names are not supported.

BFILES are read-only, so you cannot modify them. Your DBA makes sure that a given BFILE exists and that Oracle has read permissions on it. The underlying operating system maintains file integrity.

BFILES do not participate in transactions, are not recoverable, and cannot be replicated. The maximum number of open BFILES is set by the Oracle initialization parameter SESSION_MAX_OPEN_FILES, which is system dependent.

BLOB Data Type
You use the BLOB data type to store large binary objects in the database, inline or out-of-line. Every BLOB variable stores a locator, which points to a large binary object.

BLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package DBMS_LOB can be committed or rolled back. BLOB locators can span transactions (for reads only), but they cannot span sessions.

CLOB Data Type
You use the CLOB data type to store large blocks of character data in the database, inline or out-of-line. Both fixed-width and variable-width character sets are supported. Every CLOB variable stores a locator, which points to a large block of character data.

CLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package DBMS_LOB can be committed or rolled back. CLOB locators can span transactions (for reads only), but they cannot span sessions.

NCLOB Data Type
You use the NCLOB data type to store large blocks of NCHAR data in the database, inline or out-of-line. Both fixed-width and variable-width character sets are supported. Every NCLOB variable stores a locator, which points to a large block of NCHAR data.

NCLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package DBMS_LOB can be committed or rolled back. NCLOB locators can span transactions (for reads only), but they cannot span sessions.

User-Defined PL/SQL Subtypes
A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values. Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.
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 data type, and constraint applies only to base types that can specify precision and scale or a maximum size. A default value is not permitted; see Example 3–14 on page 3-27.

Examples:

   SQL> DECLARE
   2     SUBTYPE BirthDate IS DATE NOT NULL; -- Based on DATE type
   3     SUBTYPE Counter IS NATURAL;       -- Based on NATURAL subtype
   4     TYPE NameList IS TABLE OF VARCHAR2(10);       
   5     SUBTYPE DutyRoster IS NameList;              -- Based on TABLE type
   6     TYPE TimeRec IS RECORD (minutes INTEGER, hours INTEGER);
   7     SUBTYPE FinishTime IS TimeRec;                -- Based on RECORD type
   8     SUBTYPE ID_Num IS employees.employee_id%TYPE; -- Based on column type
   9     BEGIN
  10    NULL;
  11    END;
  12  /

PL/SQL procedure successfully completed.

   SQL>

You can use %TYPE or %ROWTYPE to specify the base type. When %TYPE provides the data type 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 3–15 on page 3-27. For more information, see Using the %TYPE Attribute on page 2-12 and Using the %ROWTYPE Attribute on page 2-15.

Using Subtypes

After defining a subtype, you can declare items of that type. The subtype name indicates the intended use of the variable. You can constrain a user-defined subtype when declaring variables of that type. For example:
Subtypes can increase reliability by detecting out-of-range values. Example 3–11 restricts the subtype `pinteger` to storing integers in the range -9..9. When the program tries to store a number outside that range in a `pinteger` variable, PL/SQL raises an exception.

**Example 3–11 Using Ranges with Subtypes**

```sql
SQL> DECLARE
2    v_sqlerrm VARCHAR2(64);
3
4  SUBTYPE pinteger IS PLS_INTEGER RANGE -9..9;
5    y_axis pinteger;
6
7  PROCEDURE p (x IN pinteger) IS
8    BEGIN
9      DBMS_OUTPUT.PUT_LINE (x);
10    END p;
11
12  BEGIN
13    y_axis := 9;
14    p(10);
15  EXCEPTION
16    WHEN OTHERS THEN
17      v_sqlerrm := SUBSTR(SQLERRM, 1, 64);
18      DBMS_OUTPUT.PUT_LINE('Error: ' || v_sqlerrm);
19    END;
20  /
Error: ORA-06502: PL/SQL: numeric or value error
PL/SQL procedure successfully completed.
SQL>
```

Type Compatibility with Subtypes

An unconstrained subtype is interchangeable with its base type. Example 3–12 assigns the value of `amount` to `total` without conversion.
**Example 3–12  Type Compatibility with the NUMBER Data Type**

```sql
SQL> DECLARE
2    SUBTYPE Accumulator IS NUMBER;
3    amount  NUMBER(7,2);
4    total   Accumulator;
5  BEGIN
6    amount := 10000.50;
7    total  := amount;
8  END;
9  /
PL/SQL procedure successfully completed.

SQL>
```

Different subtypes are interchangeable if they have the same base type:

```sql
SQL> DECLARE
2    SUBTYPE b1 IS BOOLEAN;
3    SUBTYPE b2 IS BOOLEAN;
4    finished  b1;
5    debugging b2;
6  BEGIN
7    finished  := FALSE;
8    debugging := finished;
9  END;
10  /
PL/SQL procedure successfully completed.

SQL>
```

Different subtypes are also interchangeable if their base types are in the same data type family. For example, the value of `verb` can be assigned to `sentence`:

```sql
SQL> DECLARE
2    SUBTYPE Word IS CHAR(15);
3    SUBTYPE Text IS VARCHAR2(1500);
4    verb     Word;
5    sentence Text(150);
6  BEGIN
7    verb := 'program';
8    sentence := verb;
9  END;
10  /
PL/SQL procedure successfully completed.

SQL>
```

**Constraints and Default Values with Subtypes**

**Example 3–13** shows to assign a default value to a subtype variable.

**Example 3–13  Assigning Default Value to Subtype Variable**

```sql
SQL> DECLARE
2    SUBTYPE v_word IS VARCHAR2(10) NOT NULL;
3    verb  v_word  := 'verb';
4    noun   v_word  := 'noun';
5  BEGIN
```
PL/SQL procedure successfully completed.

SQL>

In Example 3–14, the procedure enforces the NOT NULL constraint, but not the size constraint.

Example 3–14  Subtype Constraints Inherited by Subprograms

```sql
SQL> DECLARE
2    SUBTYPE v_word IS VARCHAR2(10) NOT NULL;
3    verb    v_word       := 'run';
4    noun    VARCHAR2(10) := NULL;
5
6    PROCEDURE word_to_upper (w IN v_word) IS
7        BEGIN
8            DBMS_OUTPUT.PUT_LINE (UPPER(w));
9        END word_to_upper;
10
11  BEGIN
12      word_to_upper('more than ten characters');
13      word_to_upper(noun);
14  END;
15 /
MORE THAN TEN CHARACTERS
DECLARE
* 
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 13
SQL>
```

As Example 3–15 shows, subtypes do not inherit the column constraints NOT NULL or CHECK, but they do inherit column size constraints.

Example 3–15  Column Constraints Inherited by Subtypes

```sql
SQL> CREATE TABLE employees_temp (  
2      empid NUMBER(6) NOT NULL PRIMARY KEY,  
3      deptid NUMBER(6) CONSTRAINT c_employees_temp_deptid  
4          CHECK (deptid BETWEEN 100 AND 200),  
5      deptname VARCHAR2(30) DEFAULT 'Sales'  
6    );
Table created.
SQL>
SQL> DECLARE
2    SUBTYPE v_empid_subtype    IS employees_temp.empid%TYPE;
3    SUBTYPE v_deptid_subtype   IS employees_temp.deptid%TYPE;
4    SUBTYPE v_deptname_subtype IS employees_temp.deptname%TYPE;
5    SUBTYPE v_emprec_subtype   IS employees_temp%ROWTYPE;
6
```
PL/SQL Data Type Conversion

Sometimes it is necessary to convert a value from one data type to another. For example, to use a DATE value in a report, you must convert it to a character string. PL/SQL supports both explicit and implicit data type conversion.

For best reliability and maintainability, use explicit conversion. Implicit conversion is context-sensitive and not always predictable, and its rules might change in later software releases. Implicit conversion can also be slower than explicit conversion.

Topics:
- Explicit Conversion
- Implicit Conversion

Explicit Conversion

To explicitly convert values from one data type to another, you use built-in functions, which are described in Oracle Database SQL Language Reference. For example, to convert a CHAR value to a DATE or NUMBER value, you use the function TO_DATE or TO_NUMBER, respectively. Conversely, to convert a DATE or NUMBER value to a CHAR value, you use the function TO_CHAR.

Explicit conversion can prevent errors or unexpected results. For example:

- Using the concatenation operator (||) to concatenate a string and an arithmetic expression can produce an error, which you can prevent by using the TO_CHAR function to convert the arithmetic expression to a string before concatenation.
Relying on language settings in the database for the format of a DATE value can produce unexpected results, which you can prevent by using the TO_CHAR function and specifying the format that you want.

**Implicit Conversion**

Sometimes PL/SQL can convert a value from one data type to another automatically. This is called implicit conversion, and the data types are called compatible. When two data types are compatible, you can use a value of one type where a value of the other type is expected. For example, you can pass a numeric literal to a subprogram that expects a string value, and the subprogram receives the string representation of the number.

In Example 3–16, 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 3–16 Implicit Conversion**

```sql
SQL> DECLARE
2   start_time   CHAR(5);
3   finish_time  CHAR(5);
4   elapsed_time NUMBER(5);
5  BEGIN
6    -- Get system time as seconds past midnight:
7    SELECT TO_CHAR(SYSDATE,'SSSSS') INTO start_time FROM sys.DUAL;
8
9    -- Processing done here
10   -- Get system time again:
11   SELECT TO_CHAR(SYSDATE,'SSSSS') INTO finish_time FROM sys.DUAL;
12
13   -- Compute and report elapsed time in seconds:
14   elapsed_time := finish_time - start_time;
15   DBMS_OUTPUT.PUT_LINE ('Elapsed time: ' || TO_CHAR(elapsed_time));
16  END;
17  /
Elapsed time: 0

PL/SQL procedure successfully completed.

SQL>
```

If you select a value from a column of one data type, and assign that value to a variable of another data type, PL/SQL converts the value to the data type of the variable. This happens, for example, when you select a DATE column value into a VARCHAR2 variable.

If you assign the value of a variable of one database type to a column of another database type, PL/SQL converts the value of the variable to the data type of the column.

If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. In such cases, you must use explicit conversion.

Table 3–10 shows which implicit conversions PL/SQL can do. However:
Table 3–10 lists only data types that have different representations.

Types that have the same representation, such as PLS_INTEGER and BINARY_INTEGER, CLOB and NCLOB, CHAR and NCHAR, and VARCHAR and NVARCHAR2, can be substituted for each other.

It is your responsibility to ensure that specific values are convertible.

For example, PL/SQL can convert the CHAR value ‘02–JUN–92’ to a DATE value but cannot convert the CHAR value ‘YESTERDAY’ to a DATE value. Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value.

Regarding date, time, and interval data types:

– Conversion rules for the DATE data type also apply to the datetime data types. However, because of their different internal representations, these types cannot always be converted to each other. For details about implicit conversions between datetime datatypes, see Oracle Database SQL Language Reference.

– To implicitly convert a DATE value to a CHAR or VARCHAR2 value, PL/SQL invokes the function TO_CHAR, which returns a character string in the default date format. To get other information, such as the time or Julian date, invoke TO_CHAR explicitly with a format mask.

– When you insert a CHAR or VARCHAR2 value into a DATE column, PL/SQL implicitly converts the CHAR or VARCHAR2 value to a DATE value by invoking the function TO_DATE, which expects its parameter to be in the default date format. To insert dates in other formats, invoke TO_DATE explicitly with a format mask.

Regarding LOB data types:

– Converting between CLOB and NCLOB values can be expensive. To make clear that you intend this conversion, use the explicit conversion functions TO_CLOB and TO_NCLOB.

– Implicit conversion between CLOB values and CHAR and VARCHAR2 values, and between BLOB values and RAW values, lets you use LOB data types in most SQL and PL/SQL statements and functions. However, to read, write, and do piecewise operations on LOB values, you must use DBMS_LOB package subprograms, which are described in Oracle Database PL/SQL Packages and Types Reference.

Regarding RAW and LONG RAW data types:

– LONG RAW is supported only for backward compatibility with existing applications. For more information, see LONG and LONG RAW Data Types on page 3-14.

– When you select a RAW or LONG RAW column value into a CHAR or VARCHAR2 variable, PL/SQL must convert the internal binary value to a character value. PL/SQL does this by returning each binary byte of RAW or LONG RAW data as a pair of characters. Each character represents the hexadecimal equivalent of a nibble (half a byte). For example, PL/SQL returns the binary byte 11111111 as the pair of characters ‘FF’. The function RAWTOHEX does the same conversion.

– Conversion is also necessary when you insert a CHAR or VARCHAR2 value into a RAW or LONG RAW column. Each pair of characters in the variable must
represent the hexadecimal equivalent of a binary byte; otherwise, PL/SQL raises an exception.

- When a LONG value appears in a SQL statement, PL/SQL binds the LONG value as a VARCHAR2 value. However, if the length of the bound VARCHAR2 value exceeds the maximum width of a VARCHAR2 column (4,000 bytes), Oracle converts the bind type to LONG automatically, and then issues an error message because you cannot pass LONG values to a SQL function.

<table>
<thead>
<tr>
<th>Table 3–10 Possible Implicit PL/SQL Data Type Conversions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>From:</strong></td>
</tr>
<tr>
<td>BLOB</td>
</tr>
<tr>
<td>CHAR</td>
</tr>
<tr>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
</tr>
<tr>
<td>LONG</td>
</tr>
<tr>
<td>NUMBER</td>
</tr>
<tr>
<td>PLS_INTEGER</td>
</tr>
<tr>
<td>RAW</td>
</tr>
<tr>
<td>UROWID</td>
</tr>
<tr>
<td>VARCHAR2</td>
</tr>
</tbody>
</table>
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. 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 13-17.

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  Simple IF-THEN Statement

    SQL> DECLARE
    2         sales  NUMBER(8,2) := 10100;
    3         quota NUMBER(8,2) := 10000;
    4         bonus NUMBER(6,2);
    5         emp_id NUMBER(6) := 120;
    6     BEGIN
    7         IF sales > (quota + 200) THEN
    8             bonus := (sales - quota)/4;
    9         END IF;
    10     UPDATE employees SET salary =
    11         salary + bonus
    12         WHERE employee_id = emp_id;
    13     END;
    14     END;
    15     /

    PL/SQL procedure successfully completed.

    SQL>

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.

Example 4–2  Using a Simple IF-THEN-ELSE Statement

```
SQL> DECLARE
2    sales  NUMBER(8,2) := 12100;
3    quota  NUMBER(8,2) := 10000;
4    bonus  NUMBER(6,2);
5    emp_id NUMBER(6) := 120;
6  BEGIN
7    IF sales > (quota + 200) THEN
8      bonus := (sales - quota)/4;
9    ELSE
10       bonus := 50;
11    END IF;
12
13    UPDATE employees
14      SET salary = salary + bonus
15        WHERE employee_id = emp_id;
16  END;
17  /
```

PL/SQL procedure successfully completed.

SQL>

IF statements can be nested. Example 4–3 shows nested IF-THEN-ELSE statements.

Example 4–3  Nested IF-THEN-ELSE Statements

```
SQL> DECLARE
2    sales  NUMBER(8,2) := 12100;
3    quota  NUMBER(8,2) := 10000;
4    bonus  NUMBER(6,2);
5    emp_id NUMBER(6) := 120;
6  BEGIN
7    IF sales > (quota + 200) THEN
8      bonus := (sales - quota)/4;
9    ELSE
10       IF sales > quota THEN
11          bonus := (sales - quota)/4;
12       ELSE
13          bonus := 0;
14       END IF;
15    END IF;
16
17    UPDATE employees
18      SET salary = salary + bonus
19        WHERE employee_id = emp_id;
20  END;
21  /
```

PL/SQL procedure successfully completed.

SQL>
Testing Conditions (IF and CASE Statements)

Using the IF-THEN-ELSIF Statement

Sometimes you want to choose between several alternatives. You can use the keyword ELSIF (not ELIF 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-ELSIF Statement

```sql
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;
/
```

PL/SQL procedure successfully completed.

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 UPDATE statement.

Another example of an IF-THEN-ELSE statement is Example 4–5.

Example 4–5 Extended IF-THEN Statement

```sql
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');
  END IF;
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 UPDATE statement.

Another example of an IF-THEN-ELSE statement is Example 4–5.
Using PL/SQL Control Structures

4-5

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.

You can rewrite the code in Example 4–5 using the CASE statement, as shown in Example 4–6.

**Example 4–6 Simple CASE Statement**

```sql
SQL> DECLARE
  2    grade CHAR(1);
  3  BEGIN
  4    grade := 'B';
  5
  6    CASE grade
  7      WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
  8      WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
  9      WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
 10      WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
 11      WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
 12      ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
 13    END CASE;
 14  END;
 15  /
Very Good
PL/SQL procedure successfully completed.
SQL>
```

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 data type 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.

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 2-40.

### 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, and its WHEN clauses contain search conditions that yield Boolean values, not expressions that can yield a value of any type.

The searched CASE statement in Example 4–7 is logically equivalent to the simple CASE statement in Example 4–6.

**Example 4–7  Searched CASE Statement**

```sql
SQL> DECLARE
2    grade CHAR(1);
3    BEGIN
4        grade := 'B';
5    CASE
6        WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
7        WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
8        WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
9        WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
10       WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
11       ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
12    END CASE;
13 END;
14 /
Very Good
```

PL/SQL procedure successfully completed.
In both Example 4–7 and Example 4–6, the ELSE clause can be replaced by an EXCEPTION part. Example 4–8 is logically equivalent to Example 4–7.

**Example 4–8  Using EXCEPTION instead of ELSE Clause in CASE Statement**

```sql
SQL> DECLARE
2    grade CHAR(1);
3  BEGIN
4    grade := 'B';
5
6    CASE
7      WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
8      WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
9      WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
10      WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
11      WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
12    END CASE;
13
14    EXCEPTION
15      WHEN CASE_NOT_FOUND THEN
16        DBMS_OUTPUT.PUT_LINE('No such grade');
17    END;
18  /
Very Good
```

PL/SQL procedure successfully completed.

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;
```

```
IF condition1 THEN statement1;
  ELSIF condition2 THEN statement2;
  ELSIF 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 RETURN Statement on page 8-4).

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 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 13-88.

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 Example 4–9 shows.

For the syntax of the EXIT statement, see EXIT Statement on page 13-51.

**Example 4–9  EXIT Statement**

```
SQL> DECLARE
2    x NUMBER := 0;
3  BEGIN
4    LOOP
5      DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
6      x := x + 1;
7      IF x > 3 THEN
8        EXIT;
9      END IF;
10    END LOOP;
11    -- After EXIT, control resumes here
12    DBMS_OUTPUT.PUT_LINE ('After loop:  x = ' || TO_CHAR(x));
13  END;
14  /
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
```
After loop:  x = 4
PL/SQL procedure successfully completed.

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 4–10.

The EXIT-WHEN statement replaces a statement of the form IF ... THEN ... EXIT. Example 4–10 is logically equivalent to Example 4–9.

For the syntax of the EXIT-WHEN statement, see EXIT Statement on page 13-51.

Example 4–10  Using an EXIT-WHEN Statement

```sql
SQL> DECLARE
2    x NUMBER := 0;
3  BEGIN
4       LOOP
5         DBMS_OUTPUT.PUT_LINE
6           ('Inside loop:  x = ' || TO_CHAR(x));
7       x := x + 1;
8
9       EXIT WHEN x > 3;
10  END LOOP;
11
12  -- After EXIT statement, control resumes here
13  DBMS_OUTPUT.PUT_LINE
14    ('After loop:  x = ' || TO_CHAR(x));
15  END;
16 /
17
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4

PL/SQL procedure successfully completed.

SQL>
```

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 4–11.

A CONTINUE statement cannot cross a subprogram or method boundary.

For the syntax of the CONTINUE statement, see CONTINUE Statement on page 13-35.
Example 4–11 CONTINUE Statement

SQL> DECLARE
2    x NUMBER := 0;
3  BEGIN
4    LOOP -- After CONTINUE statement, control resumes here
5      DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
6      x := x + 1;
7     IF x < 3 THEN
8              CONTINUE;
9     END IF;
10    DBMS_OUTPUT.PUT_LINE ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
11    EXIT WHEN x = 5;
12  END LOOP;
13  END;
14 / 
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

PL/SQL procedure successfully completed.

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 4–12 is logically equivalent to Example 4–11.

A CONTINUE-WHEN statement cannot cross a subprogram or method boundary.

For the syntax of the CONTINUE-WHEN statement, see CONTINUE Statement on page 13-35.

Example 4–12 CONTINUE-WHEN Statement

SQL> DECLARE
2    x NUMBER := 0;
3  BEGIN
4    LOOP -- After CONTINUE statement, control resumes here
5      DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
6      x := x + 1;
7     IF x < 3 THEN
8              CONTINUE;
9     END IF;
10    DBMS_OUTPUT.PUT_LINE ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
11    EXIT WHEN x = 5;
12  END LOOP;
13  END;
14 / 
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

PL/SQL procedure successfully completed.
Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements)

```sql
2  x NUMBER := 0;
3  BEGIN
4    LOOP -- After CONTINUE statement, control resumes here
5      DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
6      x := x + 1;
7      CONTINUE WHEN x < 3;
8      DBMS_OUTPUT.PUT_LINE ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
9      EXIT WHEN x = 5;
10    END LOOP;
11    DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
12  END;
13 /
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

PL/SQL procedure successfully completed.
```

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 4–13. 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 labeled loop and exit any enclosed loops.

**Example 4–13  Labeled Loops**

```sql
2  s  PLS_INTEGER := 0;
3  i  PLS_INTEGER := 0;
4  j  PLS_INTEGER;
5  BEGIN
6    LOOP <--outer_loop>>
7      i := i + 1;
8      j := 0;
9      LOOP <--inner_loop>>
10        j := j + 1;
11        s := s + i * j; -- Sum several products
12      EXIT inner_loop WHEN (j > 5);
13      EXIT outer_loop WHEN ((i * j) > 15);
14    END LOOP outer_loop;
15  END;  
```

4-12  Oracle Database PL/SQL Language Reference
Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements)

Using PL/SQL Control Structures

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 1–12 on page 1-16 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 (lower_bound .. upper_bound). The number of iterations is known before the loop is entered. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If lower_bound equals upper_bound, the loop body is executed once.

As Example 4–14 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 4–14  Simple FOR-LOOP Statement

```sql
SQL> BEGIN
  2    FOR i IN 1..3 LOOP
  3      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  4    END LOOP;
  5  END;
  6  /
  7
  8
  9
PL/SQL procedure successfully completed.

SQL>
```

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 4–15  Reverse FOR-LOOP Statement

```sql
SQL> BEGIN
  2    FOR i IN REVERSE 1..3 LOOP
  3      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  4    END LOOP;
  5  END;
  6  /
  7
  8
PL/SQL procedure successfully completed.

SQL>
```

Inside a FOR loop, the counter can be read but cannot be changed. For example:

```sql
SQL> BEGIN
  2    FOR i IN 1..3 LOOP
  3      IF i < 3 THEN
  4         DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  5      ELSE
  6         i := 2;
  7      END IF;
  8    END LOOP;
  9    /
 10    i := 2;
```

*ERROR at line 6:
ORA-06550: line 6, column 8:
PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored

SQL>
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 6-18.

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 either literals, variables, or expressions, but they 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.

Example 4–16  Several Types of FOR-LOOP Bounds

```sql
SQL> DECLARE
2  first  INTEGER := 1;
3  last   INTEGER := 10;
4  high   INTEGER := 100;
5  low    INTEGER := 12;
6  BEGIN
7    -- Bounds are numeric literals:
8    FOR j IN -5..5 LOOP
9      NULL;
10    END LOOP;
11
12    -- Bounds are numeric variables:
13    FOR k IN REVERSE first..last LOOP
14      NULL;
15    END LOOP;
16
17    -- Lower bound is numeric literal,
18    -- Upper bound is numeric expression:
19    FOR step IN 0..(TRUNC(high/low) * 2) LOOP
20      NULL;
21    END LOOP;
22  END;
23  /
PL/SQL procedure successfully completed.
SQL>
```

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 Data Types 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.

**Example 4–17** assigns today's date to elements 5, 10, and 15 of an index-by table.

**Example 4–17  Changing the Increment of the Counter in a FOR-LOOP Statement**

```sql
SQL> DECLARE
  2    TYPE DateList IS TABLE OF DATE INDEX BY PLS_INTEGER;
  3    dates DateList;
  4  BEGIN
  5      FOR j IN 1..3 LOOP
  6        dates(j*5) := SYSDATE;
  7      END LOOP;
  8    END;
  9  /

PL/SQL procedure successfully completed.

SQL>
```

**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 4–18**.

**Example 4–18  Specifying a LOOP Range at Run Time**

```sql
SQL> CREATE TABLE temp (  
  2    emp_no NUMBER,
  3    email_addr VARCHAR2(50)  
  4  );
Table created.

SQL> DECLARE
  2    emp_count NUMBER;
  3  BEGIN
  4    SELECT COUNT(employee_id) INTO emp_count
  5      FROM employees;
  6    FOR i IN 1..emp_count LOOP
  7      INSERT INTO temp
  8        VALUES(i, 'to be added later');
  9    END LOOP;
 10  END;
 11  /

PL/SQL procedure successfully completed.

SQL>
```

If the lower bound of a loop range is larger than the upper bound, the loop body is not executed and control passes to the next statement, as **Example 4–19** shows.

**Example 4–19  FOR-LOOP with Lower Bound > Upper Bound**

```sql
SQL> CREATE OR REPLACE PROCEDURE p
```
Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements)

Using PL/SQL Control Structures

```sql
(393x34) 4-17

2 (limit IN INTEGER) IS
3 BEGIN
4  FOR i IN 2..limit LOOP
5    DBMS_OUTPUT.PUT_LINE
6      ('Inside loop, limit is ' || i);
7  END LOOP;
8
9  DBMS_OUTPUT.PUT_LINE
10    ('Outside loop, limit is ' || TO_CHAR(limit));
11 END;
12 /

Procedure created.

SQL> BEGIN
2    p(3);
3  END;
4  / Inside loop, limit is 2
Inside loop, limit is 3
Outside loop, limit is 3

PL/SQL procedure successfully completed.

SQL> BEGIN
2    p(1);
3  END;
4  /
Outside loop, limit is 1

PL/SQL procedure successfully completed.

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, as Example 4–20 shows.

Example 4–20 Referencing Counter Variable Outside Loop

```sql
SQL> BEGIN
2    FOR i IN 1..3 LOOP
3      DBMS_OUTPUT.PUT_LINE
4        ('Inside loop, i is ' || TO_CHAR(i));
5    END LOOP;
6
7    DBMS_OUTPUT.PUT_LINE
8        ('Outside loop, i is ' || TO_CHAR(i));
9  END;
10 /

('Outside loop, i is ' || TO_CHAR(i));

* ERROR at line 8:
ORA-06550: line 8, column 39:
PLS-00201: identifier 'I' must be declared
ORA-06550: line 7, column 3:
PL/SQL: Statement ignored
```
Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements)

You need not declare the loop counter because it is implicitly declared as a local variable of type INTEGER. It is safest not to give a loop variable the same name as an existing variable, because the local declaration hides the global declaration, as Example 4–21 shows.

**Example 4–21  Using Existing Variable as Loop Variable**

```sql
SQL> DECLARE
2    i NUMBER := 5;
3  BEGIN
4    FOR i IN 1..3 LOOP
5      DBMS_OUTPUT.PUT_LINE
6        ('Inside loop, i is ' || TO_CHAR(i));
7    END LOOP;
8
9    DBMS_OUTPUT.PUT_LINE
10        ('Outside loop, i is ' || TO_CHAR(i));
11  END;
12  /
Inside loop, i is 1
Inside loop, i is 2
Inside loop, i is 3
Outside loop, i is 5

PL/SQL procedure successfully completed.
```

To reference the global variable in Example 4–21, you must use a label and dot notation, as in Example 4–22.

**Example 4–22  Referencing Global Variable with Same Name as Loop Counter**

```sql
SQL> <<main>>
2  DECLARE
3    i NUMBER := 5;
4  BEGIN
5    FOR i IN 1..3 LOOP
6      DBMS_OUTPUT.PUT_LINE
7        ('local: ' || TO_CHAR(i) || ', global: ' || TO_CHAR(main.i));
8    END LOOP;
9   END main;
10  /
local: 1, global: 5
local: 2, global: 5
local: 3, global: 5

PL/SQL procedure successfully completed.
```

The same scope rules apply to nested FOR loops. In Example 4–23, the inner and outer loop counters have the same name, and the inner loop uses a label and dot notation to reference the counter of the outer loop.

**Example 4–23  Referencing Outer Counter with Same Name as Inner Counter**

```sql
SQL> BEGIN
```
Controlling Loop Iterations (\texttt{LOOP}, \texttt{EXIT}, and \texttt{CONTINUE} Statements)

```
2  <<outer_loop>>
3  FOR i IN 1..3 LOOP
4    \texttt{<<inner_loop>>}
5    FOR i IN 1..3 LOOP
6      IF outer_loop.i = 2 THEN
7        \texttt{DBMS_OUTPUT.PUT_LINE(outer_loop.i || ' inner: ' \texttt{inner_loop.i});}
8      END IF;
9    END LOOP inner_loop;
10  END LOOP outer_loop;
11  END;
12 /
13 outer: 2 inner: 1
14 outer: 2 inner: 2
15 outer: 2 inner: 3
```

PL/SQL procedure successfully completed.

SQL>

\textbf{Using the EXIT Statement in a FOR Loop}

The \texttt{EXIT} statement lets a \texttt{FOR} loop complete early. In \textbf{Example 4–24}, the loop normally executes ten times, but as soon as the \texttt{FETCH} statement fails to return a row, the loop completes no matter how many times it has executed.

\textbf{Example 4–24  EXIT in a FOR LOOP}

```
SQL> DECLARE
2     v_employees employees%ROWTYPE;
3     CURSOR c1 is SELECT * FROM employees;
4  BEGIN
5    OPEN c1;
6    -- Fetch entire row into v_employees record:
7    FOR i IN 1..10 LOOP
8      FETCH c1 INTO v_employees;
9        EXIT WHEN c1%NOTFOUND;
10      -- Process data here
11    END LOOP;
12    CLOSE c1;
13  END;
14 /
```

PL/SQL procedure successfully completed.

SQL>

Suppose you must exit early from a nested \texttt{FOR} loop. To complete not only the current loop, but also any enclosing loop, label the enclosing loop and use the label in an \texttt{EXIT} statement as shown in \textbf{Example 4–25}. To complete the current iteration of the labeled loop and exit any enclosed loops, use a label in a \texttt{CONTINUE} statement.

\textbf{Example 4–25  EXIT with a Label in a FOR LOOP}

```
SQL> DECLARE
2     v_employees employees%ROWTYPE;
3     CURSOR c1 is SELECT * FROM employees;
4  BEGIN
5    OPEN c1;
6
```
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.”

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.

Example 4–26 Simple GOTO Statement

SQL> DECLARE
  2    p  VARCHAR2(30);
  3    n  PLS_INTEGER := 37;
  4  BEGIN
  5    FOR j in 2..ROUND(SQRT(n)) LOOP
  6      IF n MOD j = 0 THEN
  7        p := ' is not a prime number';
  8        GOTO print_now;
  9      END IF;
 10    END LOOP;
 11  END;
12    p := ' is a prime number';
13
14    <<print_now>>
15    DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
16    END;
17  /
37 is a prime number

PL/SQL procedure successfully completed.

SQL>

A label can appear only before a block (as in Example 4–22) or before a statement (as in Example 4–26), not within a statement, as in Example 4–27.

Example 4–27 Incorrect Label Placement

SQL> DECLARE
2    done  BOOLEAN;
3  BEGIN
4    FOR i IN 1..50 LOOP
5      IF done THEN
6        GOTO end_loop;
7      END IF;
8      <<end_loop>>
9      END LOOP;
10  END;
11  /
* ERROR at line 9:
ORA-06550: line 9, column 3:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
 begin case declare exit for goto if loop mod null raise return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge pipe purge

SQL>

To correct Example 4–27, add a NULL statement, as in Example 4–28.

Example 4–28 Using a NULL Statement to Allow a GOTO to a Label

SQL> DECLARE
2    done  BOOLEAN;
3  BEGIN
4    FOR i IN 1..50 LOOP
5      IF done THEN
6        GOTO end_loop;
7      END IF;
8      <<end_loop>>
9          NULL;
10      END LOOP;
11  END;
12  /

PL/SQL procedure successfully completed.
A GOTO statement can branch to an enclosing block from the current block, as in Example 4-29.

**Example 4-29 Using a GOTO Statement to Branch to an Enclosing Block**

```sql
SQL> DECLARE
2    v_last_name  VARCHAR2(25);
3    v_emp_id     NUMBER(6) := 120;
4  BEGIN
5    <<get_name>>
6    SELECT last_name INTO v_last_name
7      FROM employees
8        WHERE employee_id = v_emp_id;
9
10    BEGIN
11      DBMS_OUTPUT.PUT_LINE (v_last_name);
12      v_emp_id := v_emp_id + 5;
13      IF v_emp_id < 120 THEN
14        GOTO get_name;
15      END IF;
16    END;
17  END;
18  /
Weiss
SQL>
```

PL/SQL procedure successfully completed.

The GOTO statement branches to the first enclosing block in which the referenced label appears.

**GOTO Statement Restrictions**

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement, or sub-block.
- 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, either use the RETURN statement or have GOTO 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.

The GOTO statement in Example 4-30 branches into an IF statement, causing an error.

**Example 4-30 GOTO Statement Cannot Branch into IF Statement**

```sql
SQL> DECLARE
2    valid BOOLEAN := TRUE;
3  BEGIN
```
Using the NULL Statement

The NULL statement does nothing except pass control to the next statement. Some languages refer to such an instruction as a no-op (no operation). For its syntax, see NULL Statement on page 13-93.

In Example 4–31, the NULL statement emphasizes that only salespersons receive commissions.

Example 4–31 Using the NULL Statement to Show No Action

```sql
DECLARE
  v_job_id VARCHAR2(10);
  v_emp_id NUMBER(6) := 110;
BEGIN
  SELECT job_id INTO v_job_id
  FROM employees
  WHERE employee_id = v_emp_id;
  IF v_job_id = 'SA_REP' THEN
    UPDATE employees
    SET commission_pct = commission_pct * 1.2;
  ELSE
    NULL;  -- Employee is not a sales rep
  END IF;
END;
/
```

PL/SQL procedure successfully completed.

The NULL statement is a handy way to create placeholders and stub subprograms. In Example 4–32, the NULL statement lets you compile this subprogram, then fill in the real body later. Using the NULL statement might raise an unreachable code warning if warnings are enabled. See Overview of PL/SQL Compile-Time Warnings on page 11-19.
**Example 4–32 Using NULL as a Placeholder When Creating a Subprogram**

SQL> CREATE OR REPLACE PROCEDURE award_bonus
2    (emp_id NUMBER,
3     bonus NUMBER) AS
4  BEGIN    -- Executable part starts here
5      NULL;  -- Placeholder
6    -- (raises "unreachable code" if warnings enabled)
7  END award_bonus;
8  /

Procedure created.

SQL>

You can use the NULL statement to indicate that you are aware of a possibility, but that no action is necessary. In Example 4–33, the NULL statement shows that you have chosen not to take any action for unnamed exceptions.

**Example 4–33 Using the NULL Statement in WHEN OTHER Clause**

SQL> CREATE OR REPLACE FUNCTION f
2    (a INTEGER,
3     b INTEGER)
4    RETURN INTEGER
5  AS
6  BEGIN
7    RETURN (a/b);
8  EXCEPTION
9    WHEN ZERO_DIVIDE THEN
10      ROLLBACK;
11    WHEN OTHERS THEN
12      NULL;
13  END;
14  /

Function created.

SQL>

See Example 1–16, "Creating a Standalone PL/SQL Procedure" on page 1-18.
This chapter explains how to create and use PL/SQL collection and record variables. These composite variables have internal components that you can treat as individual variables. You can pass composite variables to subprograms as parameters.

To create a collection or record variable, you first define a collection or record type, and then you declare a variable of that type. In this book, collection or record means both the type and the variables of that type, unless otherwise noted.

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

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

Collections topics:
- Understanding PL/SQL Collection Types
- Choosing PL/SQL Collection Types
- Defining Collection Types
- Declaring Collection Variables
- Initializing and Referencing Collections
- Referencing Collection Elements
- Assigning Values to Collections
- Comparing Collections
- Using Multidimensional Collections
- Using Collection Methods
- Avoiding Collection Exceptions

Records topics:
- Defining and Declaring Records
- Using Records as Subprogram Parameters and Function Return Values
- Assigning Values to Records

Understanding PL/SQL Collection Types

PL/SQL has three collection types, whose characteristics are summarized in Table 5–1.
Understanding PL/SQL Collection Types

**Unbounded** means that, theoretically, there is no limit to the number of elements in the collection. Actually, there are limits, but they are very high—for details, see Referencing Collection Elements on page 5-12.

**Dense** means that the collection has no gaps between elements—every element between the first and last element is defined and has a value (which can be NULL).

A collection that is created in a PL/SQL block (with the syntax in Collection on page 13-22) is available only in that block. A nested table type or varray type that is created at schema level (with the CREATE TYPE Statement on page 14-65) is stored in the database, and you can manipulate it with SQL statements.

A collection has only one dimension, but you can model a multidimensional collection by creating a collection whose elements are also collections. For examples, see Using Multidimensional Collections on page 5-19.

Topics:
- Understanding Associative Arrays (Index-By Tables)
- Understanding Nested Tables
- Understanding Variable-Size Arrays (Varrays)

### Understanding Associative Arrays (Index-By Tables)

An associative array (also called an index-by table) is a set of key-value pairs. Each key is unique, and is used to locate the corresponding value. The key can be either an integer or a string.

Using a key-value pair for the first time adds that pair to the associative array. Using the same key with a different value changes the value.

**Example 5–1** declares an associative array that is indexed by a string, populates it, and prints it.

#### Example 5–1  Declaring and Using an Associative Array

```sql
SQL> DECLARE
2    -- Associative array indexed by string:
3        
4    TYPE population IS TABLE OF NUMBER  -- Associative array type
5        INDEX BY VARCHAR2(64);   
6        
7    city_population population;  -- Associative array variable
8    i VARCHAR2(64);          
```
BEGIN
  -- Add new elements to associative array:
  city_population('Smallville') := 2000;
  city_population('Midland')    := 750000;
  city_population('Megalopolis') := 1000000;

  -- Change value associated with key 'Smallville':
  city_population('Smallville') := 2001;

  -- Print associative array:
  i := city_population.FIRST;
  WHILE i IS NOT NULL LOOP
    DBMS_Output.PUT_LINE ('Population of ' || i || ' is ' || TO_CHAR(city_population(i)));
    i := city_population.NEXT(i);
  END LOOP;
END;
/

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

PL/SQL procedure successfully completed.

Like a database table, an associative array holds a data set of arbitrary size, and you can access its elements without knowing their positions in the array. An associative array does not need the disk space or network operations of a database table, but an associative array cannot be manipulated by SQL statements (such as INSERT and DELETE).

An associative array is intended for temporary data storage. To make an associative array persistent for the life of a database session, declare the associative array (the type and the variable of that type) in a package, and assign values to its elements in the package body.

**Globalization Settings Can Affect String Keys of Associative Arrays**

Associative arrays that are indexed by strings can be affected by globalization settings such as NLS_SORT, NLS_COMP, and NLS_DATE_FORMAT.

As Example 5–1 shows, string keys of an associative array are not stored in creation order, but in sorted order. Sorted order is determined by the initialization parameters NLS_SORT and NLS_COMP. If you change the setting of either of these parameters after populating an associated array, and then try to traverse the array, you might get an error when using a collection method such as NEXT or PRIOR. If you must change these settings during your session, set them back to their original values before performing further operations on associative arrays that are indexed by strings.

When you declare an associative array that is indexed by strings, the string type in the declaration must be VARCHAR2 or one of its subtypes. However, the key values with which you populate the array can be of any data type that can be converted to VARCHAR2 by the TO_CHAR function.
If you use key values of data types other than VARCHAR2 and its subtypes, be sure that these key values will be consistent and unique even if the settings of initialization parameters change. For example:

- Do not use `TO_CHAR(SYSDATE)` as a key value. If the `NLS_DATE_FORMAT` initialization parameter setting changes, `array_element(TO_CHAR(SYSDATE))` might return a different result.

- Two different `NVARCHAR2` values might be converted to the same `VARCHAR2` value (containing question marks instead of certain national characters), in which case `array_element(national_string1)` and `array_element(national_string2)` would refer to the same element.

- Two `CHAR` or `VARCHAR2` values that differ only in case, accented characters, or punctuation characters might also be considered the same if the value of the `NLS_SORT` initialization parameter ends in `_CI` (case-insensitive comparisons) or `_AI` (accent- and case-insensitive comparisons).

When you pass an associative array as a parameter to a remote database using a database link, the two databases can have different globalization settings. When the remote database uses a collection method such as `FIRST` or `NEXT`, it uses its own character order, which might be different from the order where the collection originated. If character set differences mean that two keys that were unique are not unique on the remote database, the program raises a `VALUE_ERROR` exception.

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

Understanding Nested Tables

Conceptually, a nested table is like a one-dimensional array with an arbitrary number of elements.

Within the database, a nested table is a column type that holds a set of values. The database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. These subscripts give you array-like access to individual rows.

A nested table differs from an array in these important ways:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically (however, a maximum limit is imposed—see Referencing Collection Elements on page 5-12).

- An array is always dense (that is, it always has consecutive subscripts). A nested array is dense initially, but it can become sparse, because you can delete elements from it.

Figure 5–1 shows the important differences between a nested table and an array.
Understanding Variable-Size Arrays (Varrays)

A variable-size array (varray) is an item of the data type VARRAY. A varray has a maximum size, which you specify in its type definition. A varray can contain a varying number of elements, from zero (when empty) to the maximum size. A varray index has a fixed lower bound of 1 and an extensible upper bound. To access an element of a varray, you use standard subscripting syntax.

Figure 5–2 shows a varray named Grades, which has maximum size 10 and contains seven elements. The current upper bound for Grades is 7, but you can increase it to the maximum of 10. Grades(n) references the nth element of Grades.

Choosing PL/SQL Collection Types

If you already have code or business logic that uses another language, you can usually translate the array and set types of that language directly to PL/SQL collection types. For example:

- Arrays in other languages become varrays in PL/SQL.
- Sets and bags in other languages become nested tables in PL/SQL.
- Hash tables and other unordered tables in other languages become associative arrays in PL/SQL.

When you are writing original code or designing original business logic, consider the characteristics of each collection type, and choose the best one for each situation.

Topics:

- Choosing Between Nested Tables and Associative Arrays
- Choosing Between Nested Tables and Varrays

See Also: Table 5–1, "Characteristics of PL/SQL Collection Types”
A nested table can be stored in a database column; therefore, you can use a nested table to simplify SQL operations in which you join a single-column table with a larger table. An associative array cannot be stored in the database.

An associative array is appropriate for the following:

- A relatively small lookup table, where the collection can be constructed in memory each time a subprogram is invoked or a package is initialized
- Passing collections to and from the database server

PL/SQL automatically converts between host arrays and associative arrays that use numeric key values. The most efficient way to pass collections to and from the database server is to set up data values in associative arrays, and then use those associative arrays with bulk constructs (the FORALL statement or BULK COLLECT clause).

### Choosing Between Nested Tables and Varrays

Varrays are a good choice when:

- The number of elements is known in advance.
- The elements are usually accessed sequentially.

When stored in the database, varrays keep their ordering and subscripts.

A varray is stored as a single object. If a varray is less than 4 KB, it is stored inside the table of which it is a column; otherwise, it is stored outside the table but in the same tablespace.

You must store or retrieve all elements of a varray at the same time, which is appropriate when operating on all the elements at once. However, this might be impractical for large numbers of elements.

Nested tables are a good choice when:

- Index values are not consecutive.
- There is no set number of index values.
- You must delete or update some elements, but not all elements at once.
- You would create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

Nested table data is stored in a separate store table, a system-generated database table. When you access a nested table, the database joins the nested table with its store table. This makes nested tables suitable for queries and updates that 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 a collection, you define a collection type and then declare variables of that type.

You can define a collection type either at schema level, inside a package, or inside a PL/SQL block. A collection type created at schema level is a standalone stored type.
You create it with the `CREATE TYPE` statement. It is stored in the database until you drop it with the `DROP TYPE` statement.

A collection type created inside a package is a **packaged type**. It is stored in the database until you drop the package with the `DROP PACKAGE` statement.

A type created inside a PL/SQL block is available only inside that block, and is stored in the database only if that block is nested within a standalone or packaged subprogram.

Collections follow the same scoping and instantiation rules as other types and variables. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

You can define `TABLE` and `VARRAY` types in the declarative part of any PL/SQL block, subprogram, or package using a `TYPE` definition.

For nested tables and varrays declared within PL/SQL, the element type of the table or varray can be any PL/SQL data type except `REF CURSOR`.

When defining a `VARRAY` type, you must specify its maximum size with a positive integer. In the following example, you define a type that stores up to 366 dates:

```plsql
DECLARE
    TYPE Calendar IS VARRAY(366) OF DATE;
END;
```

Associative arrays let you insert elements using arbitrary key values. The keys need not be consecutive.

The key data type can be `PLS_INTEGER`, `VARCHAR2`, or one of `VARCHAR2` subtypes `VARCHAR`, `STRING`, or `LONG`.

You must specify the length of a `VARCHAR2`-based key, except for `LONG` which is equivalent to declaring a key type of `VARCHAR2(32760)`.

The types `RAW`, `LONG RAW`, `ROWID`, `CHAR`, and `CHARACTER` are not allowed as keys for an associative array. The `LONG` and `LONG RAW` data types are supported only for backward compatibility; see `LONG and LONG RAW Data Types` on page 3-14 for more information.

An initialization clause is not allowed. There is no constructor notation for associative arrays. When you reference an element of an associative array that uses a `VARCHAR2`-based key, you can use other types, such as `DATE` or `TIMESTAMP`, 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**

```plsql
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**

```sql
DECLARE
    TYPE nested_type IS TABLE OF VARCHAR2(30);
    TYPE varray_type IS VARRAY(5) OF INTEGER;
    TYPE assoc_array_num_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    TYPE assoc_array_str_type IS TABLE OF VARCHAR2(32) INDEX BY PLS_INTEGER;
    TYPE assoc_array_str_type2 IS TABLE OF VARCHAR2(32) INDEX BY VARCHAR2(64);
    v1 nested_type;
    v2 varray_type;
    v3 assoc_array_num_type;
    v4 assoc_array_str_type;
    v5 assoc_array_str_type2;
BEGIN
    -- an arbitrary number of strings can be inserted v1
    v1 := nested_type('Shipping','Sales','Finance','Payroll');
    v2 := varray_type(1, 2, 3, 4, 5); -- Up to 5 integers
    v3(99) := 10; -- Just start assigning to elements
    v3(7) := 100; -- Subscripts can be any integer values
    v4(42) := 'Smith'; -- Just start assigning to elements
    v4(54) := 'Jones'; -- Subscripts can be any integer values
    v5('Canada') := 'North America';
    -- Just start assigning to elements
    v5('Greece') := 'Europe';
    -- Subscripts can be string values
END;
/
```

As shown in Example 5–4, you can use %TYPE to specify the data type of a previously declared collection, so that changing the definition of the collection automatically updates other variables that depend on the number of elements or the element type.

**Example 5–4 Declaring Collections with %TYPE**

```sql
DECLARE
    TYPE few_depts IS VARRAY(10) OF VARCHAR2(30);
    TYPE many_depts IS VARRAY(100) OF VARCHAR2(64);
    some_depts few_depts;

    /* If the type of some_depts changes from few_depts to many_depts,
       local_depts and global_depts will use the same type
       when this block is recompiled */

    local_depts some_depts%TYPE;
    global_depts some_depts%TYPE;
BEGIN
```

See Also:
- Collection on page 13-22
- CREATE TYPE Statement on page 14-65
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**

```sql
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**

```sql
DECLARE
  good_employees personnel.staff_list;
BEGIN
  good_employees := personnel.staff_list(100, 103, 107);
  personnel.award_bonuses (good_employees);
END;
/
```

You can also specify a collection type in the RETURN clause of a function specification. To specify the element type, you can use %TYPE, which provides the data type of a variable or database column. Also, you can use %ROWTYPE, which provides the rowtype of a cursor or database table. See Example 5–7 and Example 5–8.

**Example 5–7  Specifying Collection Element Types with %TYPE and %ROWTYPE**

```sql
DECLARE
  -- Nested table type that can hold an arbitrary number
  -- of employee IDs.
  -- The element type is based on a column from the EMPLOYEES table.
  -- You need not know whether the ID is a number or a string.
  -- Declare a cursor to select a subset of columns.
  -- Declare an Array type that can hold information
  -- about 10 employees.
  -- The element type is a record that contains all the same
```
Initializing and Referencing Collections

-- 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));
DECLARE 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;
DECLARE v_employees EmpList := EmpList(100, 150, 160, 200);
BEGIN
  v_employees(3) := NULL; -- assigning NULL raises an exception
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**

```plsql
DECLARE
  -- In the varray, put an upper limit on the number of elements
  TYPE dnames_var IS VARRAY(20) OF VARCHAR2(30);
  dept_names dnames_var;
BEGIN
  -- Because dnames is declared as VARRAY(20),
  -- you can put up to 10 elements in the constructor
  dept_names := dnames_var('Shipping','Sales','Finance','Payroll');
END;
/
```

Unless you impose the **NOT NULL** constraint in the type declaration, you can pass null
elements to a constructor as in Example 5–12.

**Example 5–12 Collection Constructor Including Null Elements**

```plsql
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30);
  dept_names dnames_tab;
  TYPE dnamesNoNulls_type IS TABLE OF VARCHAR2(30) NOT NULL;
BEGIN
  dept_names := dnames_tab('Shipping', NULL, 'Finance', NULL);
  -- If dept_names were of type dnamesNoNulls_type,
  -- you could not include null values in the constructor
END;
/
```

You can initialize a collection in its declaration, which is a good programming practice,
as shown in Example 5–13. In this case, you can invoke the collection's **EXTEND**
method to add elements later.

**Example 5–13 Combining Collection Declaration and Constructor**

```plsql
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**

```plsql
DECLARE
  TYPE dnames_var IS VARRAY(20) OF VARCHAR2(30);
  dept_names dnames_var;
```
BEGIN
  IF dept_names IS NULL THEN
    DBMS_OUTPUT.PUT_LINE
    ('Before initialization, the varray is null.');
    -- While the varray is null, you cannot check its COUNT attribute.
    -- DBMS_OUTPUT.PUT_LINE
    -- ('It has ' || dept_names.COUNT || ' elements.');
  ELSE
    DBMS_OUTPUT.PUT_LINE
    ('Before initialization, the varray is not null.');
  END IF;

  dept_names := dnames_var(); -- initialize empty varray
  IF dept_names IS NULL THEN
    DBMS_OUTPUT.PUT_LINE
    ('After initialization, the varray is null.');
  ELSE
    DBMS_OUTPUT.PUT_LINE
    ('After initialization, the varray is not null.');
    DBMS_OUTPUT.PUT_LINE
    ('It has ' || dept_names.COUNT || ' elements.');
  END IF;
END;
/

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:

\[ \text{collection_name} \left( \text{subscript} \right) \]

where \text{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..\text{size_limit}, where you specify the limit in the declaration (\text{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.

\textbf{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
    -- invoke function to retrieve
    -- element identified by subscript (key)
    DBMS_OUTPUT.PUT_LINE
    ('Sum of the first ' || TO_CHAR(n) || ' multiples of ' ||
     TO_CHAR(m) || ' is ' || TO_CHAR(get_sum_multiples (m, sn)(n)));
END;
/

Assigning Values to Collections

One collection can be assigned to another by an INSERT, UPDATE, FETCH, or SELECT statement, an assignment statement, or a subprogram call. You can assign the value of an expression to a specific element in a collection using the syntax:

collection_name (subscript) := expression;

where expression yields a value of the type specified for elements in the collection type definition.

You can use operators such as SET, MULTISET UNION, MULTISET INTERSECT, and MULTISET EXCEPT to transform nested tables as part of an assignment statement.

Assigning a value to a collection element can raise exceptions, for example:

- If the subscript is NULL or is not convertible to the right data type, PL/SQL raises the predefined exception VALUE_ERROR. Usually, the subscript must be an integer. Associative arrays can also be declared to have VARCHAR2 subscripts.
- If the subscript refers to an uninitialized element, PL/SQL raises `SUBSCRIPT_BEYOND_COUNT`.
- If the collection is atomically null, PL/SQL raises `COLLECTION_IS_NULL`.

For more information about collection exceptions, see Avoiding Collection Exceptions on page 5-29, Example 5–38 on page 5-29, and Predefined PL/SQL Exceptions on page 11-4.

Example 5–17 shows that collections must have the same data type for an assignment to work. Having the same element type is not enough.

**Example 5–17  Data Type Compatibility for Collection Assignment**

```sql
DECLARE
  TYPE last_name_typ IS VARRAY(3) OF VARCHAR2(64);
  TYPE surname_typ IS VARRAY(3) OF VARCHAR2(64);
  -- These first two variables have the same data type.
  group1 last_name_typ := last_name_typ('Jones','Wong','Marceau');
  group2 last_name_typ := last_name_typ('Klein','Patsos','Singh');
  -- This third variable has a similar declaration,
  -- but is not the same type.
  group3 surname_typ := surname_typ('Trevisi','Macleod','Marquez');
BEGIN
  -- Allowed because they have the same data type
  group1 := group2;
  -- Not allowed because they have different data types
  -- group3 := group2; -- raises an exception
END;
```

If you assign an atomically null nested table or varray to a second nested table or varray, the second collection must be reinitialized, as shown in Example 5–18. In the same way, assigning the value `NULL` to a collection makes it atomically null.

**Example 5–18  Assigning a Null Value to a Nested Table**

```sql
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30);
  -- This nested table has some values
  dept_names dnames_tab :=
    dnames_tab('Shipping','Sales','Finance','Payroll');
  -- This nested table is not initialized ("atomically null").
  empty_set dnames_tab;
BEGIN
  -- At first, the initialized variable is not null.
  if dept_names IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('OK, at first dept_names is not null.');
  END IF;
  -- Then assign a null nested table to it.
  dept_names := empty_set;
  -- Now it is null.
  if dept_names IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('OK, now dept_names has become null.' );
  END IF;
  -- Use another constructor to give it some values.
  dept_names := dnames_tab('Shipping','Sales','Finance','Payroll');
END;
```

5-14  Oracle Database PL/SQL Language Reference
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**

```plsql
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 MULTISET UNION nt4; -- (1,2,3,1,2,4)
  print_nested_table(answer);
  answer := nt1 MULTISET UNION nt3; -- (1,2,3,2,3,1,3)
  print_nested_table(answer);
  answer := nt1 MULTISET UNION DISTINCT nt3; -- (1,2,3)
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT nt3; -- (3,2,1)
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT DISTINCT nt3; -- (3,2,1)
  print_nested_table(answer);
  answer := SET(nt3); -- (2,3,1)
  print_nested_table(answer);
  answer := nt3 MULTISET EXCEPT nt2; -- (3)
  print_nested_table(answer);
  answer := nt3 MULTISET EXCEPT DISTINCT nt2; -- ()
  print_nested_table(answer);
END;
/
```

Example 5–20 shows an assignment to a VARRAY of records with an assignment statement.

**Example 5–20 Assigning Values to VARRAYs with Complex Data Types**

```plsql
DECLARE
  TYPE emp_name_rec is RECORD {
    firstname employees.first_name%TYPE,
    lastname employees.last_name%TYPE,
    hiredate employees.hire_date%TYPE
  };
```

Assigning Values to Collections

-- 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 Data Types

DECLARE
    TYPE emp_name_rec is RECORD {
        firstname    employees.first_name%TYPE,
        lastname     employees.last_name%TYPE,
        hiredate     employees.hire_date%TYPE
    };

    -- Table type that can hold information about employees
    TYPE EmpList_tab IS TABLE OF emp_name_rec;
    SeniorSalespeople EmpList_tab;

    -- Declare a cursor to select a subset of columns.
    CURSOR c1 IS SELECT first_name, last_name, hire_date
    FROM employees;
    EndCounter NUMBER := 10;
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv EmpCurTyp;

    BEGIN
    OPEN emp_cv FOR SELECT first_name, last_name, hire_date
    FROM employees
WHERE job_id = 'SA_REP' ORDER BY hire_date;

FETCH emp_cv BULK COLLECT INTO SeniorSalespeople;
CLOSE emp_cv;

-- for this example, display a maximum of ten employees
IF SeniorSalespeople.LAST > 0 THEN
  IF SeniorSalespeople.LAST < 10 THEN
    EndCounter := SeniorSalespeople.LAST;
  END IF;
  FOR i in 1..EndCounter LOOP
    DBMS_OUTPUT.PUT_LINE
    (SeniorSalespeople(i).lastname || ', ' ||
     SeniorSalespeople(i).firstname || ', ' ||
     SeniorSalespeople(i).hiredate);
  END LOOP;
END IF;
END;
/

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 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 you have not used a constructor.
  IF members IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not NULL');
  END IF;
END;
/
Example 5–23 shows that nested tables can be compared for equality or inequality. They cannot be ordered, because there is no greater than or less than comparison.

**Example 5–23  Comparing Two Nested Tables**

DECLARE
    TYPE dnames_tab IS TABLE OF VARCHAR2(30);
    dept_names1 dnames_tab :=
        dnames_tab('Shipping','Sales','Finance','Payroll');
    dept_names2 dnames_tab :=
        dnames_tab('Sales','Finance','Shipping','Payroll');
    dept_names3 dnames_tab :=
        dnames_tab('Sales','Finance','Payroll');
BEGIN
    -- You can use = or !=, but not < or >.
    -- These 2 are equal even though members are in different order.
    IF dept_names1 = dept_names2 THEN
        DBMS_OUTPUT.PUT_LINE
        (dept_names1 and dept_names2 have the same members.);
    END IF;
    IF dept_names2 != dept_names3 THEN
        DBMS_OUTPUT.PUT_LINE
        (dept_names2 and dept_names3 have different members.);
    END IF;
END;

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
Using Multidimensional Collections

Although a collection has only one dimension, you can model a multidimensional collection by creating a collection whose elements are also collections. For example, you can create a nested table of varrays, a varray of varrays, a varray of nested tables, and so on.

When creating a nested table of nested tables as a column in SQL, check the syntax of the `CREATE TABLE` statement to see how to define the storage table.

Example 5–25, Example 5–26, and Example 5–27 are some examples showing the syntax and possibilities for multilevel collections.

**Example 5–25  Multilevel VARRAY**

```plsql
DECLARE
    TYPE t1 IS VARRAY(10) OF INTEGER;
    TYPE nt1 IS VARRAY(10) OF t1; -- multilevel varray type
    va t1 := t1(2,3,5);
    -- initialize multilevel varray
    nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
    i INTEGER;
    va1 t1;
BEGIN
    -- multilevel access
    i := nva(2)(3); -- i will get value 73
    DBMS_OUTPUT.PUT_LINE('I = ' || i);
    -- add a new varray element to nva
    nva.EXTEND;
    -- replace inner varray elements
    nva(4) := 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**

```plsql
DECLARE
    TYPE tbl1 IS TABLE OF VARCHAR2(20);
    TYPE Ntbl1 IS TABLE OF tbl1; -- table of table elements
```
Using Collection Methods

A collection method is a built-in PL/SQL subprogram that returns information about a collection or operates on a collection. Collection methods make collections easier to use, and make your applications easier to maintain.

You invoke a collection method using dot notation. For detailed syntax, see Collection Method Call on page 13-27.

You cannot invoke a collection method from a SQL statement.

The only collection method that you can use with an empty collection is EXISTS; all others raise the exception COLLECTION_IS_NULL.

Topics:
- Checking If a Collection Element Exists (EXISTS Method)
- Counting the Elements in a Collection (COUNT Method)
Checking If a Collection Element Exists (EXISTS Method)

EXISTS \((n)\) returns TRUE if the \(n\)th element in a collection exists; otherwise, it 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**

```sql
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;
/
```

**Note:** You cannot use EXISTS with an associative array.

Counting the Elements in a Collection (COUNT Method)

COUNT returns the current number of elements in a collection. It is useful when you do not know how many elements a collection contains. For example, when you fetch a column of data into a nested table, 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 usually equals LAST. However, 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.
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;
/

Checking the Maximum Size of a Collection (LIMIT Method)

LIMIT returns the maximum number of elements that a collection can have. If the collection has no maximum size, LIMIT returns NULL.

Example 5–30  Checking the Maximum Size of a Collection with LIMIT

DECLARE
    TYPE dnames_var IS VARRAY(7) OF VARCHAR2(30);
    dept_names dnames_var :=
        dnames_var('Shipping','Sales','Finance','Payroll');
BEGIN
    DBMS_OUTPUT.PUT_LINE('dept_names has ' || dept_names.COUNT || ' elements now');
    DBMS_OUTPUT.PUT_LINE('dept_names''s type can hold a maximum of ' || dept_names.LIMIT || ' elements');
    DBMS_OUTPUT.PUT_LINE('The maximum number you can use with ' || dept_names.EXTEND() || ' is ' || (dept_names.LIMIT - dept_names.COUNT));
END;
/

Finding the First or Last Collection Element (FIRST and LAST Methods)

For a collection indexed by integers, FIRST and LAST return the first and last (smallest and largest) index numbers.

For an associative array indexed by strings, FIRST and LAST return the lowest and highest key values. If the NLS_COMP initialization parameter is set to ANSI, the order is based on the sort order specified by the NLS_SORT initialization parameter.

If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same value.

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.');
  END IF;
END;
/

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').
Using Collection Methods

Example 5–32  Using PRIOR and NEXT to Access Collection Elements

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('The element after #2 is #' || n.NEXT(2));
  DBMS_OUTPUT.PUT_LINE('The element before #2 is #' || n.PRIOR(2));
  n.DELETE(3);-- Delete an element to show how NEXT can handle gaps.
  DBMS_OUTPUT.PUT_LINE('Now the element after #2 is #' || n.NEXT(2));
  IF n.PRIOR(n.FIRST) IS NULL THEN
    DBMS_OUTPUT.PUT_LINE
      ('Can''t get PRIOR of the first element or NEXT of the last.');
  END IF;
END;
/

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.');</x.COUNT || ' elements.');
    END;
    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.

**Note:** You cannot use **TRIM** with an associative array.

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**

```sql
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).
    -- **TRIM**ming 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 these 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.

**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 collection increases as the number of elements in the collection increases. If you delete the entire collection, or delete all elements individually, all of the memory used to store elements of that collection is freed.

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, you can assign values to the elements.
  words := WordList('1st', '2nd', '3rd'); -- 3 elements created
  -- Any expression that returns a VARCHAR2(5) is valid.
  words(3) := words(1) || '+2';
  BEGIN
    words(3) := 'longer than 5 characters'; -- Raises VALUE_ERROR
    -- The assigned value is too long.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words('B') := 'dunno'; -- Raises VALUE_ERROR
    -- The subscript (B) of a nested table must be an integer.
    -- Note: Also, NULL is not allowed as a subscript.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words(0) := 'zero'; -- Raises SUBSCRIPT_OUTSIDE_LIMIT
    -- Subscript 0 is outside the allowed subscript range.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words(4) := 'maybe'; -- Raises SUBSCRIPT_BEYOND_COUNT
    -- The subscript (4) exceeds the number of elements in the table.
    -- To add 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 11-17. For information about the use of SQLERRM with exception handling, see Retrieving the Error Code and Error Message on page 11-15.

The following list summarizes when a given exception is raised.

<table>
<thead>
<tr>
<th>Collection Exception</th>
<th>Raised when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLECTION_IS_NULL</td>
<td>you try to operate on an atomically null collection.</td>
</tr>
</tbody>
</table>
Avoiding Collection Exceptions

<table>
<thead>
<tr>
<th>Collection Exception</th>
<th>Raised when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_DATA_FOUND</td>
<td>a subscript designates an element that was deleted, or a nonexistent element of an associative array.</td>
</tr>
<tr>
<td>SUBSCRIPT_BEYOND_COUNT</td>
<td>a subscript exceeds the number of elements in a collection.</td>
</tr>
<tr>
<td>SUBSCRIPT_OUTSIDE_LIMIT</td>
<td>a subscript is outside the allowed range.</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>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.</td>
</tr>
</tbody>
</table>

See Also: 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.

**Example 5–39  How Invalid Subscripts are Handled with DELETE(n)**

```sql
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, if you invoke the packaged procedure in **Example 5–40**, the second procedure call fails, because the packaged and local `VARRAY` types are incompatible despite their identical definitions.

**Example 5–40  Incompatibility Between Package and Local Collection Types**

```sql
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;
```
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 13-104.

Example 5–42 and Example 5–42 illustrate record type declarations.

Example 5–41  Declaring and Initializing a Simple Record Type

```
DECLARE
    TYPE DeptRecTyp IS RECORD (
        deptid NUMBER(4) NOT NULL := 99,
        dname departments.department_name%TYPE,
        loc departments.location_id%TYPE,
        region regions%ROWTYPE );
    dept_rec DeptRecTyp;
BEGIN
    dept_rec.dname := 'PURCHASING';
END;
/
```

Example 5–42  Declaring and Initializing Record Types

```
DECLARE
    -- Declare a record type with 3 fields.
    TYPE rec1_t IS RECORD (field1 VARCHAR2(16), field2 NUMBER, field3 DATE);
    -- For any fields declared NOT NULL, you must supply 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);
```
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**

```plsql
DECLARE
  -- Best: use %ROWTYPE instead of specifying each column.
  -- Use <cursor>%ROWTYPE instead of <table>%ROWTYPE because you 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.

To declare a record that represents a row in a database table, without listing the columns, use the `%ROWTYPE` attribute.

Your code keeps working even after columns are added to the table. If you want to represent a subset of columns in a table, or columns from different tables, you can define a view or declare a cursor to select the right columns and do any necessary joins, and then apply `%ROWTYPE` to the view or cursor.
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**

```sql
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
  nth_highest_salary (3);
END;
/```

Like scalar variables, user-defined records can be declared as the formal parameters of subprograms, as in **Example 5–45**.

**Example 5–45  Using a Record as Parameter to a Procedure**

```sql
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
  raise_salary (EmpRecTyp (1001, 5000));
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**

```plsql
DECLARE
    TYPE TimeTyp IS RECORD ( minutes SMALLINT, hours SMALLINT );
    TYPE MeetingTyp IS RECORD (      
        day     DATE,         -- nested record
        time_of TimeTyp,   -- nested record
        dept    departments%ROWTYPE,  -- nested record representing a table row
        place   VARCHAR2(20),
        purpose VARCHAR2(50) );
    meeting MeetingTyp;
    seminar MeetingTyp;
BEGIN
    -- Can assign one nested record to another
    -- if they are of the same data type
    seminar.time_of := meeting.time_of;
END;
/```

Such assignments are allowed even if the containing records have different data types.

**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**

```plsql
DECLARE
    TYPE RecordTyp IS RECORD (field1 NUMBER,      
                               field2 VARCHAR2(32) DEFAULT 'something');
    rec1 RecordTyp;
    rec2 RecordTyp;
BEGIN
    -- At first, rec1 has the values you 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:

```plsql
emp_info.last_name := 'Fields';
```

Values are assigned separately to each field of a record in Example 5–47. You cannot assign a list of values to a record using an assignment statement. There is no constructor-like notation for records.

You can assign values to all fields at once only if you assign a record to another record with the same data type. 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));
depth1_info DeptRec1;
depth2_info DeptRec2;
depth3_info DeptRec2;
BEGIN
-- Not allowed; different data types,
-- even though fields are the same.
--      dept1_info := dept2_info;
-- This assignment is OK because the records have the same type.
    dept2_info := dept3_info;
END;
/

You can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have the same data types:

DECLARE
    TYPE RecordTyp IS RECORD (last employees.last_name%TYPE,
                               id employees.employee_id%TYPE);
    CURSOR c1 IS SELECT last_name, employee_id FROM employees;
-- Rec1 and rec2 have different types,
-- but because rec2 is based on a %ROWTYPE,
-- you can assign it to rec1 as long as they have
-- the right number of fields and
-- the fields have the right data types.
    rec1 RecordTyp;
    rec2 c1%ROWTYPE;
BEGIN
    SELECT last_name, employee_id INTO rec2
    FROM employees WHERE ROWNUM < 2;
    rec1 := rec2;
    DBMS_OUTPUT.PUT_LINE
    (’Employee #' || rec1.id || ' = ' || rec1.last);
END;
/

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;
    DBMS_OUTPUT.PUT_LINE
    (’Employee #' || rec1.id || ' = ' || rec1.last);
END;
Assigning Values to Records

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 data types. 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.

```sql
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 you 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 data types.

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**

```sql
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**

```sql
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);  
END;  
ROLLBACK;
```
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**

```sql
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
  -- separate 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
```
```sql
(underpaid(i).last_name || ' makes ' || underpaid(i).salary);
END LOOP;
-- You can also bring in just some of the table columns.
-- Here you get the first and last names of 10 arbitrary employees.
SELECT first_name, last_name
  BULK COLLECT INTO some_names
  FROM employees
  WHERE ROWNUM < 11;
FOR i IN some_names.FIRST .. some_names.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
    ('Employee = ' || some_names(i).first_name || ' ' || some_names(i).last_name);
  END LOOP;
END;
/```
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 database 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.');
END;
/

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 about 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 5-37.
For more information about assigning values to PL/SQL variables, see Assigning SQL Query Results to PL/SQL Variables on page 2-27.

---

**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**

The database is transaction oriented; that is, the database 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, the database makes all the changes permanent or undoes them all. If your program fails in the middle of a transaction, the database 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**

The queries in Example 6–4 invoke a SQL function (COUNT).

---

**Example 6–4  Invoking the SQL COUNT Function in PL/SQL**

```sql
SQL> DECLARE
  2  job_count NUMBER;
  3  emp_count NUMBER;
  4  BEGIN
  5  SELECT COUNT(DISTINCT job_id)
  6    INTO job_count
  7    FROM employees;
  8  
  9  SELECT COUNT(*)
10    INTO emp_count
11    FROM employees;
12  END;
13 /

PL/SQL procedure successfully completed.
```

SQL>
SQL Pseudocolumns

PL/SQL recognizes the SQL pseudocolumns CURRVAL, LEVEL, NEXTVAL, ROWID, and ROWNUM. However, there are limitations on the use of pseudocolumns, including the restriction on the use of some pseudocolumns in assignments or conditional tests. For more information, 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

```sql
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.
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. Implicit cursors are called **SQL cursors**. 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 explicit cursors for queries that return more than one row.

**Topics:**
- SQL Cursors (Implicit)
- Explicit Cursors

SQL Cursors (Implicit)
SQL cursors are managed automatically by PL/SQL. You need not write code to handle these cursors. However, you can track information about the execution of an SQL cursor through its attributes.

**Topics:**
- Attributes of SQL Cursors
- Guidelines for Using Attributes of SQL Cursors
Attributes of SQL Cursors

SQL 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 the database opens the SQL cursor, its 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 12-14.

Topics:

- %FOUND Attribute: Has a DML Statement Changed Rows?
- %ISOPEN Attribute: Always FALSE for SQL 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

```sql
CREATE TABLE dept_temp AS SELECT * FROM departments;
DECLARE
department NUMBER(4) := 270;
BEGIN
DELETE FROM dept_temp WHERE department_id = department;
IF SQL%FOUND THEN  -- delete succeeded
    INSERT INTO dept_temp VALUES (270, 'Personnel', 200, 1700);
END IF;
END;
/
```

%ISOPEN Attribute: Always FALSE for SQL Cursors The database 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

```sql
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 SQL Cursors**

When using attributes of SQL cursors, consider the following:

- 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 SQL 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.
Managing Cursors in PL/SQL

- 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
```sql
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 a value to a cursor or use it in an expression. Cursors and variables follow the same scoping rules. Naming cursors after database tables is possible but not recommended.

A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be `IN` parameters; they supply values in the query, but do not return any values from the query. You cannot impose the 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.

Example 6–9 Declaring a Cursor
```sql
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:

```sql
DECLARE
  CURSOR c1 IS
      SELECT employee_id, last_name, job_id, salary
      FROM employees
      WHERE salary > 2000;
BEGIN
  OPEN c1;
END;
```

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

```sql
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;
```
Managing Cursors in PL/SQL

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**

```plsql
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**

```plsql
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 12-17. In Example 6–13, you bulk-fetch from a cursor into two collections.

**Example 6–13  Fetching Bulk Data with a Cursor**

```plsql
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;
    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**

```plsql
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**

```plsql
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**

```plsql
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;
```
Managing Cursors in PL/SQL

Using Static SQL

6-15

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

<table>
<thead>
<tr>
<th>Point in Time</th>
<th>%FOUND Value</th>
<th>%ISOPEN Value</th>
<th>%NOTFOUND Value</th>
<th>%ROWCOUNT Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before OPEN</td>
<td>exception</td>
<td>FALSE</td>
<td>exception</td>
<td>exception</td>
</tr>
<tr>
<td>After OPEN</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>0</td>
</tr>
</tbody>
</table>
Querying Data with PL/SQL

PL/SQL lets you perform queries and access individual fields or entire rows from the result set. In traditional database programming, you process query results using an internal data structure called a cursor. In most situations, PL/SQL can manage the cursor for you, so that code to process query results is straightforward and compact. This section explains how to process both simple queries where PL/SQL manages everything, and complex queries where you interact with the cursor.

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.

---

Table 6–1 (Cont.) Cursor Attribute Values

<table>
<thead>
<tr>
<th>Point in Time</th>
<th>%FOUND Value</th>
<th>%ISOPEN Value</th>
<th>%NOTFOUND Value</th>
<th>%ROWCOUNT Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before first FETCH</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>After first FETCH</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>1</td>
</tr>
<tr>
<td>Before each successive FETCH except last</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>1</td>
</tr>
<tr>
<td>After each successive FETCH except last</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>data dependent</td>
</tr>
<tr>
<td>Before last FETCH</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>data dependent</td>
</tr>
<tr>
<td>After last FETCH</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>data dependent</td>
</tr>
<tr>
<td>Before CLOSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>data dependent</td>
</tr>
<tr>
<td>After CLOSE</td>
<td>exception</td>
<td>FALSE</td>
<td>exception</td>
<td>exception</td>
</tr>
</tbody>
</table>

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.
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 must bring a large quantity of data into local PL/SQL variables, rather than looping through a result set one row at a time, you can use the `BULK COLLECT` clause. When you query only certain columns, you can store all the results for each column in a separate collection variable. When you query all the columns of a table, you can store the entire result set in a collection of records, which makes it convenient to loop through the results and refer to different columns. See Example 6–13, "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 must loop only 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 need not be declared in advance. It is a `%ROWTYPE` record whose field names match the column names from the query, and that exists only during the loop. When you use expressions rather than explicit column names, use column aliases so that you can refer to the corresponding values inside the loop.

**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:

■ SQL Cursor FOR LOOP
■ Explicit Cursor FOR LOOP

SQL 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.

```plsql
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 13-88.

Explicit Cursor FOR LOOP

If you must reference the same query from different parts of the same subprogram, you can declare a cursor that specifies the query, and process the results using a FOR loop.

```plsql
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;
/
```

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 in 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 these 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 is 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
    --...
Using Subqueries

-- subquery returns all the rows in descending order of salary
-- main query returns just the top 10 highest-paid employees
BEGIN
  SELECT * FROM
    (SELECT last_name, salary
     FROM employees ORDER BY salary DESC, last_name
     ORDER BY salary DESC, last_name
     WHERE ROWNUM < 11)
  ORDER BY salary DESC, last_name
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Above-average salary: ' || person.last_name);
  END LOOP;
  FOR person IN c2
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Highest paid: ' || person.last_name || ' $' || person.salary);
  END LOOP;
END;
/

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 data type 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 the database 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:

```sql
DECLARE
  TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
END;
```

REF CURSOR types can be strong (with a return type) or weak (with no return type). Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with queries that return the right set of columns. Weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query. Because there is no type checking with a weak REF CURSOR, all such types are interchangeable. Instead of creating a 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.

```sql
DECLARE
  -- Strong:
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  -- Weak:
  TYPE genericcurtyp IS REF CURSOR;
  cursor1 empcurtyp;
  cursor2 genericcurtyp;
  my_cursor SYS_REFCURSOR; -- no new type needed
  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;
  dept_cv deptcurtyp; -- declare cursor variable
END;
```
To avoid declaring the same REF CURSOR type in each subprogram that uses it, you can put the REF CURSOR declaration in a package 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**

```sql
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 data type of a record variable, as shown in Example 6–25.

**Example 6–25  Using the %ROWTYPE Attribute to Provide the Data Type**

```sql
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**

```sql
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**

```sql
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 13-96.

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 13-96.

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. 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
Using Cursor Variables (REF CURSORs)

Using Static SQL

```
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
```
Using Cursor Variables (REF CURSORs)

```
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**

```sql
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**

```sql
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:

```plaintext
/* 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 multiblock 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:

```plaintext
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 \texttt{c1}, \texttt{c2}, and \texttt{c3} act normally, and you can use them for any purpose. When finished, release the cursors as follows:

\begin{verbatim}
BEGIN
  CLOSE :c1; CLOSE :c2; CLOSE :c3;
END;
/
\end{verbatim}

Avoiding Errors with Cursor Variables

If both cursor variables involved in an assignment are strongly typed, they must have exactly the same data type (not just the same return type). If one or both cursor variables are weakly typed, they can have different data types.

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 \texttt{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 \texttt{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 \texttt{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 \texttt{FOR} loop.

Example 6–34  Declaration of Cursor Variables in a Package

\begin{verbatim}
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
\end{verbatim}
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 not an 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, and cursors produced by subqueries involving the other values in the row. A single query can return a large set of related values retrieved from multiple tables. You can process the result set with nested loops that fetch first from the rows of the result set, and then from any nested cursors 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.)

The syntax of a cursor expression is:

```sql
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 reexecuted
- 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.

In **Example 6–35**, the cursor `c1` is associated with a query that includes a cursor expression. For each department in the `departments` table, the nested cursor returns the last name of each employee in that department (which it retrieves from the `employees` table).

**Example 6–35  Using a Cursor Expression**

```sql
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
```
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 a database 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 need not write extra code to prevent problems with multiple users accessing data concurrently. The database 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 the 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**

```sql
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, the database 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.

**See Also:**
- *Oracle Database Advanced Application Developer's Guide* for more information about committing transactions
- *Oracle Database Concepts* for information about distributed transactions
- *Oracle Database SQL Language Reference* for information about the COMMIT statement

### 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');
```
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.
Overview of Transaction Processing in PL/SQL

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;
/

See Also: Oracle Database SQL Language Reference for more information about the SET TRANSACTION SQL statement

How the Database Does Implicit Rollbacks

Before executing an INSERT, UPDATE, or DELETE statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint. Usually, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

The database can also roll back single SQL statements to break deadlocks. The database signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Before executing a SQL statement, the database 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
**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**

```sql
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.

**See Also:** Oracle Database SQL Language Reference for more information about the SQL statement SET TRANSACTION

**Overriding Default Locking**

By default, the database locks data structures for you automatically, which is a major strength of the 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 must 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. For example:

```sql
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 the database 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**, the database 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** 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:

```sql
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

```sql
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.

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.

See Also:
- Oracle Database Advanced Application Developer’s Guide for more
  information about lock modes
- Oracle Database SQL Language Reference for more information about
  the LOCK TABLE SQL statement

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 exception 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;
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**

```sql
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:

```sql
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**

<table>
<thead>
<tr>
<th>Main Transaction</th>
<th>Autonomous Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE proc1 IS</td>
<td>PROCEDURE proc2 IS</td>
</tr>
<tr>
<td>emp_id NUMBER; BEGIN</td>
<td>PRAGMA AUTON...</td>
</tr>
<tr>
<td>emp_id := 7788; INSERT ...</td>
<td>dept_id NUMBER;</td>
</tr>
<tr>
<td>SELECT ... proc2; DELETE ...</td>
<td>BEGIN</td>
</tr>
<tr>
<td>COMMIT;</td>
<td>dept_id := 20;</td>
</tr>
<tr>
<td>END;</td>
<td>UPDATE ...</td>
</tr>
<tr>
<td>MT begins</td>
<td>MT suspends</td>
</tr>
<tr>
<td>MT ends</td>
<td>AT begins</td>
</tr>
<tr>
<td>MT ends</td>
<td>AT ends</td>
</tr>
<tr>
<td>MT resumes</td>
<td>MT resumes</td>
</tr>
</tbody>
</table>

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 needs not 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**

```sql
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**

```sql
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**

```sql
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**

```sql
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.
When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does not roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

**Avoiding Errors with Autonomous Transactions**

To avoid some common errors, remember the following:

- If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur. The database raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

- The database initialization parameter `TRANSACTIONS` specifies the maximum number of concurrent transactions. That number might be exceeded because an autonomous transaction runs concurrently with the main transaction.

- If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception goes unhandled, the transaction is rolled back.

**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**

```sql
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 8-25. 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;

6-46  Oracle Database PL/SQL Language Reference
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;
/

**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 data types of its input and output variables.

PL/SQL provides two ways to write dynamic SQL:

- Native dynamic SQL, a PL/SQL language (that is, native) feature for building and 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 data types of the input and output variables of the dynamic SQL statement. If you do not know this information at compile time, you must use the DBMS_SQL package.

When you need both the DBMS_SQL package and native dynamic SQL, you can switch between them, using the DBMS_SQL.TO_REFCURSOR Function on page 7-7 and DBMS_SQL.TO_CURSOR_NUMBER Function on page 7-8.

**Topics:**
- When You Need Dynamic SQL
- 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.
Using Native Dynamic SQL

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 other than SELECT, without a RETURNING INTO clause, put all bind arguments in the USING clause.
- If the dynamic SQL statement is a DML statement with a RETURNING INTO clause, put in-bind arguments in the USING clause and out-bind arguments in the RETURNING INTO clause.
If the dynamic SQL statement is an anonymous PL/SQL block or a CALL statement, put all bind arguments in the USING clause.

If the dynamic SQL statement invokes a subprogram, ensure that:

- Every bind argument that corresponds to a placeholder for a subprogram parameter has the same parameter mode as that subprogram parameter (as in Example 7–1) and a data type that is compatible with that of the subprogram parameter. (For information about compatible data types, see Formal and Actual Subprogram Parameters on page 8-6.)

- No bind argument has a data type that SQL does not support (such as BOOLEAN in Example 7–2).

The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 7–3.

For syntax details of the EXECUTE IMMEDIATE statement, see EXECUTE IMMEDIATE Statement on page 13-47.

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 Data Type in Native Dynamic SQL

DECLARE
    FUNCTION f (x INTEGER)
        RETURN BOOLEAN
    AS
        BEGIN
            ...
        END f;
    dyn_stmt VARCHAR2(200);
    b1      BOOLEAN;
Using Native Dynamic SQL

BEGIN
  dyn_stmt := 'BEGIN :b := f(5); END;
  -- Fails because SQL does not support BOOLEAN data type:
  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 13-96.

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 13-68.

3. Use the CLOSE statement to close the cursor variable.

   For syntax details, see CLOSE Statement on page 13-20.

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;
  END LOOP;
END;
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 need not repeat its corresponding bind argument. All references to that placeholder name correspond to one bind argument in the USING clause.

In Example 7–5, all references to the first unique placeholder name, :x, are associated with the first bind argument in the USING clause, a, and the second unique placeholder name, :y, is associated with the second bind argument in the USING clause, b.
Example 7–5  Repeated Placeholder Names in Dynamic PL/SQL Block

CREATE PROCEDURE calc_stats (  
  w NUMBER,  
  x NUMBER,  
  y NUMBER,  
  z NUMBER )  
IS  
BEGIN  
  DBMS_OUTPUT.PUT_LINE(w + x + y + z);  
END;  
/

DECLARE  
  a NUMBER := 4;  
  b NUMBER := 7;  
  plsql_block VARCHAR2(100);  
BEGIN  
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';  
  EXECUTE IMMEDIATE plsql_block USING a, b;  
-- calc_stats(a, a, b, a)  
END;  
/

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 data type REF CURSOR, which you can use in native dynamic SQL statements.

Before passing a SQL cursor number to the DBMS_SQL.TO_REFCURSOR function, you must OPEN, PARSE, and EXECUTE it (otherwise an error occurs).

After you convert a SQL cursor number to a REF CURSOR variable, DBMS_SQL operations can access it only as the REF CURSOR variable, not as the SQL cursor number. For example, using the DBMS_SQL.IS_OPEN function to see if a converted SQL cursor number is still open causes an error.

Example 7–6 uses the DBMS_SQL.TO_REFCURSOR function to switch from the DBMS_SQL package to native dynamic SQL.

Example 7–6 Switching from DBMS_SQL Package to Native Dynamic SQL

```
CREATE OR REPLACE TYPE vc_array IS TABLE OF VARCHAR2(200);
/
CREATE OR REPLACE TYPE numlist IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE do_query_1 (
    placeholder vc_array,
    bindvars vc_array,
    sql_stmt VARCHAR2 
) IS
    TYPE curtype IS REF CURSOR;
    src_cur      curtype;
    curid        NUMBER;
    bindnames    vc_array;
    emnos        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 emnos, depts;

    -- This would cause an error because curid was converted to a REF CURSOR:
    -- isopen := DBMS_SQL.IS_OPEN(curid);

    CLOSE src_cur;
```
Using DBMS_SQL Package

7-8 Oracle Database PL/SQL Language Reference

DBMS_SQL.TO_CURSOR_NUMBER Function

The DBMS_SQL.TO_CURSOR function converts a REF CURSOR variable (either strongly or weakly typed) to a SQL cursor number, which you can pass to DBMS_SQL subprograms.

Before passing a REF CURSOR variable to the DBMS_SQL.TO_CURSOR function, you must OPEN it.

After you convert a REF CURSOR variable to a SQL cursor number, native dynamic SQL operations cannot access it.

After a FETCH operation begins, passing the DBMS_SQL cursor number to the DBMS_SQL.TO_REFCURSOR or DBMS_SQL.TO_CURSOR function causes an error.

Example 7–7 uses the DBMS_SQL.TO_CURSOR function to switch from native dynamic SQL to the DBMS_SQL package.

Example 7–7  Switching from Native Dynamic SQL to DBMS_SQL Package

```
CREATE OR REPLACE PROCEDURE do_query_2 (sql_stmt VARCHAR2) IS
  TYPE curtype IS REF CURSOR;
  src_cur  curtype;
  curid    NUMBER;
  desctab  DBMS_SQL.DESC_TAB;
  colcnt   NUMBER;
  namevar  VARCHAR2(50);
  numvar   NUMBER;
  datevar  DATE;
  empno    NUMBER := 100;
BEGIN
  -- sql_stmt := SELECT ... FROM employees WHERE employee_id = :b1';
  -- Open REF CURSOR variable:
  OPEN src_cur FOR sql_stmt USING empno;
  -- Switch from native dynamic SQL to DBMS_SQL package:
  curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
  DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);
  -- Define columns:
  FOR i IN 1 .. colcnt LOOP
    IF desctab(i).col_type = 2 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
    ELSIF desctab(i).col_type = 12 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
    -- statements
    ELSE
      DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 50);
    END IF;
  END LOOP;
  -- Fetch rows with DBMS_SQL package:
  WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
    FOR i IN 1 .. colcnt LOOP
      IF (desctab(i).col_type = 1) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
      ELSIF (desctab(i).col_type = 2) THEN
```
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 to guard against them.

To try the examples in this topic, connect to the HR schema and execute the statements in Example 7–8.

Example 7–8  Setup for SQL Injection Examples

CREATE TABLE secret_records (  
  user_name   VARCHAR2(9),  
  service_type VARCHAR2(12),  
  value       VARCHAR2(30),  
  date_created DATE);

INSERT INTO secret_records  
VALUES ('Andy', 'Waiter', 'Serve dinner at Cafe Pete', SYSDATE);

INSERT INTO secret_records  
VALUES ('Chuck', 'Merger', 'Buy company XYZ', SYSDATE);

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
- Data Type Conversion

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.
The SQL*Plus script in Example 7–9 creates a procedure that is vulnerable to statement modification and then invokes that procedure with and without statement modification. With statement modification, the procedure returns a supposedly secret record.

**Example 7–9 Procedure Vulnerable to Statement Modification**

```sql
SQL> REM Create vulnerable procedure
SQL> CREATE OR REPLACE PROCEDURE get_record
  2    (user_name IN VARCHAR2,
  3        service_type IN VARCHAR2,
  4        record OUT VARCHAR2)
  5 IS
  6    query VARCHAR2(4000);
  7  BEGIN
  8    -- Following SELECT statement is vulnerable to modification
  9    -- because it uses concatenation to build WHERE clause.
 10    query := 'SELECT value FROM secret_records WHERE user_name='''
 11    || user_name
 12    || '' AND service_type='''
 13    || service_type
 14    || '';
 15    DBMS_OUTPUT.PUT_LINE('Query: ' || query);
 16    EXECUTE IMMEDIATE query INTO record;
 17    DBMS_OUTPUT.PUT_LINE('Record: ' || record);
 18  END;
 19 /
Procedure created.

SQL> REM Demonstrate procedure without SQL injection
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
  2    record_value VARCHAR2(4000);
  3  BEGIN
  4    get_record('Andy', 'Waiter', record_value);
  5  END;
  6  /
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND service_type='Waiter'
Record: Serve dinner at Cafe Pete
PL/SQL procedure successfully completed.

SQL>
SQL> REM Example of statement modification
SQL> DECLARE
  2    record_value VARCHAR2(4000);
  3  BEGIN
  4    get_record('Anybody ' OR service_type='Merger'--',
  5    'Anything',
  6    record_value);
  7  END;
  8  /
Query: SELECT value FROM secret_records WHERE user_name='Anybody ' OR service_type='Merger'--'
```
Avoiding SQL Injection in PL/SQL

Using Dynamic SQL

service_type='Merger'--' AND service_type='Anything'  
Record: Buy company XYZ

PL/SQL procedure successfully completed.

SQL>

Statement Injection

Statement injection means that a user appends one or more new SQL statements to a dynamic SQL statement. Anonymous PL/SQL blocks are vulnerable to this technique.

The SQL*Plus script in Example 7–10 creates a procedure that is vulnerable to statement injection and then invokes that procedure with and without statement injection. With statement injection, the procedure deletes the supposedly secret record exposed in Example 7–9.

Example 7–10  Procedure Vulnerable to Statement Injection

SQL> REM Create vulnerable procedure
SQL>
CREATE OR REPLACE PROCEDURE p
  (user_name IN VARCHAR2,
   service_type IN VARCHAR2)
IS
  block VARCHAR2(4000);
BEGIN
  -- Following block is vulnerable to statement injection
  -- because it is built by concatenation.
  block :=
    'BEGIN
      DBMS_OUTPUT.PUT_LINE(''user_name: ' || user_name || ''');
      DBMS_OUTPUT.PUT_LINE(''service_type: ' || service_type || ''');
    END;';
  DBMS_OUTPUT.PUT_LINE('Block: ' || block);
  EXECUTE IMMEDIATE block;
END;
/
Procedure created.

SQL>

SQL> REM Demonstrate procedure without SQL injection
SQL>
SET SERVEROUTPUT ON;
SQL>
BEGIN
  p('Andy', 'Waiter');
END;
/
Block: BEGIN
  DBMS_OUTPUT.PUT_LINE('user_name: Andy');
  DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
END;
user_name: Andy
service_type: Waiter

PL/SQL procedure successfully completed.
Avoiding SQL Injection in PL/SQL

SQL> REM Example of statement modification
SQL> SELECT * FROM secret_records;
USER_NAME SERVICE_TYPE VALUE
--------- ------------ ------------------------------
Andy      Waiter       Serve dinner at Cafe Pete
Chuck     Merger       Buy company XYZ

2 rows selected.

SQL> BEGIN
2    p('Anybody', 'Anything');
3  DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
4  END;
5  /
Block: BEGIN
    DBMS_OUTPUT.PUT_LINE('user_name: Anybody');
    DBMS_OUTPUT.PUT_LINE('service_type: Anything');
    DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
END;
user_name: Anybody
service_type: Anything
PL/SQL procedure successfully completed.

SQL> SELECT * FROM secret_records;
USER_NAME SERVICE_TYPE VALUE
--------- ------------ ------------------------------
Andy      Waiter       Serve dinner at Cafe Pete

1 row selected.

SQL>

Data Type Conversion
A less known SQL injection technique uses NLS session parameters to modify or inject SQL statements.

A datetime or numeric value that is concatenated into the text of a dynamic SQL statement must be converted to the VARCHAR2 data type. The conversion can be either implicit (when the value is an operand of the concatenation operator) or explicit (when the value is the argument of the TO_CHAR function). This data type conversion depends on the NLS settings of the database session that executes the dynamic SQL statement. The conversion of datetime values uses format models specified in the parameters NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT, or NLS_TIMESTAMP_TZ_FORMAT, depending on the particular datetime data type. The conversion of numeric values applies decimal and group separators specified in the parameter NLS_NUMERIC_CHARACTERS.

One datetime format model is "text". The text is copied into the conversion result. For example, if the value of NLS_DATE_FORMAT is '"Month: Month', then in June, TO_CHAR(SYSDATE) returns 'Month: June'. The datetime format model can be abused as shown in Example 7-11.
Example 7-11  Procedure Vulnerable to SQL Injection Through Data Type Conversion

```sql
Example 7–11 Procedure Vulnerable to SQL Injection Through Data Type Conversion

SQL> REM Create vulnerable procedure
SQL> REM Return records not older than a month
SQL> CREATE OR REPLACE PROCEDURE get_recent_record
    | (user_name    IN  VARCHAR2,
    | service_type IN  VARCHAR2,
    | record       OUT VARCHAR2)
IS
    | query VARCHAR2(4000);
BEGIN
    -- Following SELECT statement is vulnerable to modification
    -- because it uses concatenation to build WHERE clause
    -- and because SYSDATE depends on the value of NLS_DATE_FORMAT.
    query := 'SELECT value FROM secret_records WHERE user_name='''
    | || user_name
    | ||' AND service_type='''
    | || service_type
    | ||' AND date_created='''
    | || (SYSDATE - 30)
    | || '''
    DBMS_OUTPUT.PUT_LINE('Query: ' || query);
    EXECUTE IMMEDIATE query INTO record;
    DBMS_OUTPUT.PUT_LINE('Record: ' || record);
END;
/
.
Procedure created.
.
SQL> REM Demonstrate procedure without SQL injection
SQL> SET SERVEROUTPUT ON;
SQL>
SQL> ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';
.
Session altered.
.
SQL> DECLARE
2    record_value VARCHAR2(4000);
3  BEGIN
4    get_recent_record('Andy', 'Waiter', record_value);
5  END;
6  /
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND service_type='Waiter' AND date_created>'27-MAY-2008'
Record: Serve dinner at Cafe Pete
PL/SQL procedure successfully completed.
.
SQL>
SQL> REM Example of statement modification
SQL>
SQL> ALTER SESSION SET NLS_DATE_FORMAT=''' OR service_type='Merger';
.
Session altered.
.
SQL> DECLARE
2    record_value VARCHAR2(4000);
3  BEGIN
4    get_recent_record('Anybody', 'Anything', record_value);
```
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 Explicit Format Models to Guard Against SQL Injection

Using Bind Arguments to Guard Against SQL Injection

The most effective way to make your PL/SQL code invulnerable to SQL injection attacks is to use bind arguments. The database uses the values of bind arguments exclusively and does not interpret their contents in any way. (Bind arguments also improve performance.)

The procedure in Example 7–12 is invulnerable to SQL injection because it builds the dynamic SQL statement with bind arguments (not by concatenation as in the vulnerable procedure in Example 7–9). The same binding technique fixes the vulnerable procedure shown in Example 7–10.

**Example 7–12 Using Bind Arguments to Guard Against SQL Injection**

```sql
REM Create invulnerable procedure
SQL> CREATE OR REPLACE PROCEDURE get_record_2
2    (user_name    IN  VARCHAR2,
3     service_type IN  VARCHAR2,
4     record       OUT VARCHAR2)
5  IS
6    query VARCHAR2(4000);
7  BEGIN
8    query := 'SELECT value FROM secret_records
9              WHERE user_name=:a
10              AND service_type=:b';
11
12    DBMS_OUTPUT.PUT_LINE('Query: ' || query);
13
14    EXECUTE IMMEDIATE query INTO record
15    USING user_name, service_type;
16
17    DBMS_OUTPUT.PUT_LINE('Record: ' || record);
18  END;
19  /

Procedure created.
```

SQL> REM Demonstrate procedure without SQL injection
SQL> SQL> SET SERVEROUTPUT ON;

PL/SQL procedure successfully completed.
SQL>
Avoiding SQL Injection in PL/SQL

Using Dynamic SQL

```sql
SQL>
SQL> DECLARE
2    record_value VARCHAR2(4000);
3  BEGIN
4    get_record_2('Andy', 'Waiter', record_value);
5  END;
6  /
Query: SELECT value FROM secret_records
       WHERE user_name=:a
       AND service_type=:b
Record: Serve dinner at Cafe Pete

PL/SQL procedure successfully completed.

SQL>
SQL> REM Attempt statement modification
SQL>
SQL> DECLARE
2    record_value VARCHAR2(4000);
3  BEGIN
4    get_record_2('Anybody '' OR service_type=''Merger''--',
5                 'Anything',
6                 record_value);
7  END;
8  /
Query: SELECT value FROM secret_records
       WHERE user_name=:a
       AND service_type=:b
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "HR.GET_RECORD_2", line 14
ORA-06512: at line 4

SQL>
```

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`.

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.

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, as Example 7–13 does. This prevents a malicious user from injecting text between an opening quotation mark and its corresponding closing quotation mark.
Avoiding SQL Injection in PL/SQL

Caution: Although the DBMS_ASSERT subprograms are useful in validation code, they do not replace it. For example, an input string can be a qualified SQL name (verified by DBMS_ASSERT.QUALIFIED_SQL_NAME) and still be a fraudulent password.

See Also: Oracle Database PL/SQL Packages and Types Reference for information about DBMS_ASSERT subprograms

In Example 7–13, the procedure raise_emp_salary checks the validity of the column name that was passed to it before it updates the employees table, and then the anonymous PL/SQL block invokes the procedure from both a dynamic PL/SQL block and a dynamic SQL statement.

Example 7–13 Using Validation Checks to Guard Against SQL Injection

```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;  
/
```

7-16 Oracle Database PL/SQL Language Reference
Using Explicit Format Models to Guard Against SQL Injection

If you use datetime and numeric values that are concatenated into the text of a SQL or PL/SQL statement, and you cannot pass them as bind variables, convert them to text using explicit format models that are independent from the values of the NLS parameters of the executing session. Ensure that the converted values have the format of SQL datetime or numeric literals. Using explicit locale-independent format models to construct SQL is recommended not only from a security perspective, but also to ensure that the dynamic SQL statement runs correctly in any globalization environment.

The procedure in Example 7–14 is invulnerable to SQL injection because it converts the datetime parameter value, SYSDATE – 30, to a VARCHAR2 value explicitly, using the TO_CHAR function and a locale-independent format model (not implicitly, as in the vulnerable procedure in Example 7–11).

Example 7–14 Using Explicit Format Models to Guard Against SQL Injection

```
SQL> REM Create invulnerable procedure
SQL> REM Return records not older than a month
SQL> CREATE OR REPLACE PROCEDURE get_recent_record
(user_name    IN  VARCHAR2,
 service_type IN  VARCHAR2,
 record       OUT VARCHAR2)
IS
 query VARCHAR2(4000);
BEGIN
   -- Following SELECT statement is vulnerable to modification
   -- because it uses concatenation to build WHERE clause.
   query := 'SELECT value FROM secret_records WHERE user_name='''
     || user_name
     || ''' AND service_type='''
     || service_type
     || ''' AND date_created> DATE ''
     || TO_CHAR(SYSDATE - 30,'YYYY-MM-DD')
     || '''
;
   DBMS_OUTPUT.PUT_LINE('Query: ' || query);
   EXECUTE IMMEDIATE query INTO record;
   DBMS_OUTPUT.PUT_LINE('Record: ' || record);
END;
/.
Procedure created.
.
SQL> REM Attempt statement modification
SQL> ALTER SESSION SET NLS_DATE_FORMAT=''OR service_type=''Merger'';
.
Session altered.
.
SQL> DECLARE
2    record_value VARCHAR2(4000);
3 BEGIN
4    get_recent_record('Anybody', 'Anything', record_value);
5 END;
6 /.
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
    service_type='Anything' AND date_created> DATE '2008-05-27'
```
DECLARE
  *
  ERROR at line 1:
  ORA-01403: no data found
  ORA-06512: at "SYS.GET_RECENT_RECORD", line 18
  ORA-06512: at line 4
.
SQL>
This chapter explains how to turn sets of statements into reusable subprograms. Subprograms are the building blocks of modular, maintainable applications.

Topics:
- Overview of PL/SQL Subprograms
- Subprogram Parts
- Creating Nested Subprograms that Invoke Each Other
- Declaring and Passing Subprogram Parameters
- 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 PL/SQL Function Result Cache

Overview of PL/SQL Subprograms

A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters. A subprogram can be either a procedure or a function. Typically, you use a procedure to perform an action and a function to compute and return a value.

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

A subprogram created at schema level is a standalone stored subprogram. You create it with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database until you drop it with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a packaged subprogram. It is stored in the database until you drop the package with the DROP PACKAGE statement.

A subprogram created inside a PL/SQL block is a nested subprogram. You can either declare and define it at the same time, or you can declare it first (forward declaration) and then define it later in the same block. A nested subprogram is stored in the database only if it is nested within a standalone or packaged subprogram.
See Also:

- CREATE PROCEDURE Statement on page 14-46 for more information about creating standalone stored procedures
- CREATE FUNCTION Statement on page 14-29 for more information about creating standalone stored functions
- CREATE PACKAGE Statement on page 14-39 for more information about creating standalone stored functions
- Procedure Declaration and Definition on page 13-101 for more information about creating procedures inside PL/SQL blocks
- Function Declaration and Definition on page 13-75 for more information about creating functions inside PL/SQL blocks

Subprogram Calls
A subprogram call has this form:

```plsql
subprogram_name [ (parameter [, parameter]... ) ]
```

A procedure call is a PL/SQL statement. For example:

```plsql
raise_salary(employee_id, amount);
```

A function call is part of an expression. For example:

```plsql
IF salary_ok(new_salary, new_title) THEN ... 
```

See Also: Declaring and Passing Subprogram Parameters on page 8-6 for more information about subprogram calls

Reasons to Use Subprograms
- Subprograms let you extend the PL/SQL language.
  Procedure calls are like new statements. Function calls are like new expressions and operators.
- Subprograms let you break a program into manageable, well-defined modules.
  You can use top-down design and the stepwise refinement approach to problem solving.
- Subprograms promote re-usability.
  Once tested, a subprogram can be reused in any number of applications. You can invoke PL/SQL subprograms from many different environments, so that you need not rewrite them each time you use a new language or use a new API to access the database.
- Subprograms promote maintainability.
  You can change the internal details of a subprogram without changing the other subprograms that invoke it. Subprograms are an important component of other maintainability features, such as packages and object types.
- Dummy subprograms ("stubs") let you defer the definition of procedures and functions until after you have tested the main program.
  You can design applications from the top down, thinking abstractly, without worrying about implementation details.
- Subprograms can be grouped into PL/SQL packages.
Packages make code even more reusable and maintainable, and can be used to define an API.

- You can hide the implementation details of subprograms by placing them in PL/SQL packages.

You can define subprograms in a package body without declaring their specifications in the package specification. However, such subprograms can be invoked only from inside the package. At least one statement must appear in the executable part of a subprogram. The NULL statement meets this requirement.

Subprogram Parts

A subprogram always has a name, and can have a parameter list.

Like every PL/SQL block, a subprogram has an optional declarative part, a required executable part, and an optional exception-handling part, and can specify PRAGMA AUTONOMOUS_TRANSACTION, which makes it autonomous (independent).

The declarative part of a subprogram does not begin with the keyword DECLARE, as the declarative part of a non-subprogram block does. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

The executable part of a subprogram contains statements that assign values, control execution, and manipulate data.

The exception-handling part of a subprogram contains code that handles run-time errors.

Example 8–1 declares and defines a procedure (at the same time) inside an anonymous block. The procedure has the required executable part and the optional exception-handling part, but not the optional declarative part. The executable part of the block invokes the procedure.

Example 8–1 Declaring, Defining, and Invoking a Simple PL/SQL Procedure

```sql
-- Declarative part of block begins
DECLARE
  in_string  VARCHAR2(100) := 'This is my test string.';
  out_string VARCHAR2(200);

-- Procedure declaration and definition begins
PROCEDURE double (original IN VARCHAR2,
           new_string OUT VARCHAR2)
IS
  -- Declarative part of procedure (optional) goes here
  -- Executable part of procedure begins
  BEGIN
    new_string := original || ' + ' || original;
  -- Executable part of procedure ends
  -- Exception-handling part of procedure begins
  EXCEPTION
    WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE('Output buffer not long enough.');
  -- Exception-handling part of procedure ends
  -- Procedure declaration and definition ends
-- Declarative part of block ends
-- Executable part of block begins
```
BEGIN
  double(in_string, out_string);  -- Procedure invocation
  DBMS_OUTPUT.PUT_LINE(in_string || ' - ' || out_string);
END;
-- Executable part of block ends
/

A procedure and a function have the same structure, except that:

- A function heading must include a RETURN clause that specifies the data type of the return value. A procedure heading cannot have a RETURN clause.

- A function must have at least one RETURN statement in its executable part. In a procedure, the RETURN statement is optional. For details, see RETURN Statement on page 8-4.

- Only a function heading can include the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETERMINISTIC option</td>
<td>Helps the optimizer avoid redundant function calls.</td>
</tr>
<tr>
<td>PARALLEL_ENABLED option</td>
<td>Allows the function to be used safely in slave sessions of parallel DML evaluations.</td>
</tr>
<tr>
<td>PIPELINED option</td>
<td>Returns the results of a table function iteratively.</td>
</tr>
<tr>
<td>RESULT_CACHE option</td>
<td>Stores function results in the PL/SQL function result cache.</td>
</tr>
<tr>
<td>RESULT_CACHE clause</td>
<td>Specifies the data sources on which the results of a function.</td>
</tr>
</tbody>
</table>

See Also:

- Procedure Declaration and Definition on page 13-101 for the syntax of procedure declarations and definitions

- Function Declaration and Definition on page 13-75 for the syntax of function declarations and definitions, including descriptions of the items in the preceding table

- Declaring and Passing Subprogram Parameters on page 8-6 for more information about subprogram parameters

- Using the PL/SQL Function Result Cache on page 8-27 for more information about the RESULT_CACHE option and the RESULT_CACHE clause

RETURN Statement

The RETURN statement (not to be confused with the RETURN clause, which specifies the data type of the return value of a function) immediately ends the execution of the subprogram that contains it and returns control to the caller. Execution continues with the statement following the subprogram call.

A subprogram can contain several RETURN statements. The subprogram need not end with a RETURN statement. Executing any RETURN statement completes the subprogram immediately.

In a procedure, a RETURN statement cannot contain an expression and does not return a value.

In a function, a RETURN statement must contain an expression. When the RETURN statement executes, the expression is evaluated, and its value is assigned to the
function identifier. The function identifier acts like a variable of the type specified in the RETURN clause.

The expression in a function RETURN statement can be arbitrarily complex. For example:

```sql
CREATE OR REPLACE FUNCTION half_of_square(original NUMBER) RETURN NUMBER IS
BEGIN
    RETURN (original * original)/2 + (original * 4);
END half_of_square;
/
```

A function must have at least one execution path that leads to a RETURN statement.

See Also: RETURN Statement on page 13-109 for the syntax of the RETURN statement

Example 8–2 declares and defines a function (at the same time) inside an anonymous block. The function has the optional declarative part and the required executable part, but not the optional exception-handling part. The executable part of the block invokes the function.

Example 8–2 Declaring, Defining, and Invoking a Simple PL/SQL Function

```sql
-- Declarative part of block begins
DECLARE
    -- Function declaration and definition begins
    FUNCTION square (original NUMBER) RETURN NUMBER -- RETURN clause
    AS
    -- Declarative part of function begins
    original_squared NUMBER;
    -- Declarative part of function ends
    -- Executable part of function begins
    BEGIN
        original_squared := original * original;
        RETURN original_squared; -- RETURN statement
    END;
    -- Executable part of function ends
    -- Function declaration and definition ends
    -- Declarative part of block ends
    -- Executable part of block begins
    BEGIN
        DBMS_OUTPUT.PUT_LINE(square(100)); -- Function invocation
    END;
    -- Executable part of block ends
/
```

Creating Nested Subprograms that Invoke Each Other

In a block, you can create multiple nested subprograms. If they invoke each other, you need forward declaration, because a subprogram must be declared before it can be invoked. With forward declaration, you declare a subprogram, but do not define it until after you have defined the other subprograms that invoke it. A forward declaration and its corresponding definition must appear in the same block.

The block in Example 8–3 creates two procedures that invoke each other.
Declaring and Passing Subprogram Parameters

A subprogram heading can declare formal parameters. Each formal parameter declaration can specify a mode and a default value. When you invoke the subprogram, you can pass actual parameters to it.

Topics:

- Formal and Actual Subprogram Parameters
- Specifying Subprogram Parameter Modes
- Specifying Default Values for Subprogram Parameters
- Passing Actual Subprogram Parameters with Positional, Named, or Mixed Notation

Formal and Actual Subprogram Parameters

**Formal parameters** are the variables declared in the subprogram header and referenced in its execution part. **Actual parameters** are the variables or expressions that you pass to the subprogram when you invoke it. Corresponding formal and actual parameters must have compatible data types.

A good programming practice is to use different names for formal and actual parameters, as in Example 8–4.

**Example 8–4  Formal Parameters and Actual Parameters**

```plsql
DECLARE
    emp_num NUMBER(6) := 120;
    bonus   NUMBER(6) := 100;
    merit   NUMBER(4) := 50;

    PROCEDURE raise_salary (    -- formal parameter
        emp_id NUMBER,          -- formal parameter
        amount NUMBER
    ) IS
        BEGIN
            NULL;
        END;
/
```

---

### Example 8–3  Creating Nested Subprograms that Invoke Each Other

```plsql
DECLARE
    -- Declare proc1 (forward declaration):
    PROCEDURE proc1(number1 NUMBER);

    -- Declare and define proc 2:
    PROCEDURE proc2(number2 NUMBER) IS
        BEGIN
            proc1(number2);
        END;

    -- Define proc 1:
    PROCEDURE proc1(number1 NUMBER) IS
        BEGIN
            proc2(number1);
        END;

    BEGIN
        NULL;
    END;
/
```

---

Declaring and Passing Subprogram Parameters

A subprogram heading can declare formal parameters. Each formal parameter declaration can specify a mode and a default value. When you invoke the subprogram, you can pass actual parameters to it.

Topics:

- Formal and Actual Subprogram Parameters
- Specifying Subprogram Parameter Modes
- Specifying Default Values for Subprogram Parameters
- Passing Actual Subprogram Parameters with Positional, Named, or Mixed Notation

Formal and Actual Subprogram Parameters

**Formal parameters** are the variables declared in the subprogram header and referenced in its execution part. **Actual parameters** are the variables or expressions that you pass to the subprogram when you invoke it. Corresponding formal and actual parameters must have compatible data types.

A good programming practice is to use different names for formal and actual parameters, as in Example 8–4.

**Example 8–4  Formal Parameters and Actual Parameters**

```plsql
DECLARE
    emp_num NUMBER(6) := 120;
    bonus   NUMBER(6) := 100;
    merit   NUMBER(4) := 50;

    PROCEDURE raise_salary (    -- formal parameter
        emp_id NUMBER,          -- formal parameter
        amount NUMBER
    ) IS
        BEGIN
            NULL;
        END;
/
```

---

### Example 8–3  Creating Nested Subprograms that Invoke Each Other

```plsql
DECLARE
    -- Declare proc1 (forward declaration):
    PROCEDURE proc1(number1 NUMBER);

    -- Declare and define proc 2:
    PROCEDURE proc2(number2 NUMBER) IS
        BEGIN
            proc1(number2);
        END;

    -- Define proc 1:
    PROCEDURE proc1(number1 NUMBER) IS
        BEGIN
            proc2(number1);
        END;

    BEGIN
        NULL;
    END;
/
```
UPDATE employees
SET salary = salary + amount
WHERE employee_id = emp_id;
END raise_salary;
BEGIN
raise_salary(emp_num, bonus); -- actual parameters
raise_salary(emp_num, merit + bonus); -- actual parameters
END;
/

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 data type of the actual parameter to the data type of the corresponding formal parameter before the assignment (this is why corresponding formal and actual parameters must have compatible data types). For information about implicit conversion, see Implicit Conversion on page 3-29.

A good programming practice is to avoid implicit conversion, either by using explicit conversion (explained in Explicit Conversion on page 3-28) or by declaring the variables that you intend to use as actual parameters with the same data types 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 packages and types that Oracle supplies 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 implementation of the supplied packages types.

---

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 Specifying Default Values for Subprogram Parameters on page 8-9.

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, as in Example 8–5.

Example 8–5 Using OUT Mode

```sql
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 Specifying Default Values for Subprogram Parameters on page 8-9.

The initial value of an OUT parameter is NULL; therefore, the data type of an OUT parameter cannot be a subtype defined as NOT NULL, such as 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 8–1 summarizes the characteristics of parameter modes.

<table>
<thead>
<tr>
<th>Parameter Modes</th>
<th>IN</th>
<th>OUT</th>
<th>IN OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default</td>
<td>Must be specified</td>
<td>Returns a value to the caller</td>
<td>Must be specified</td>
</tr>
<tr>
<td>Passes a value to the subprogram</td>
<td>Formal parameter acts like a constant</td>
<td>Formal parameter acts like an initialized variable</td>
<td>Formal parameter acts like a variable</td>
</tr>
<tr>
<td>Formal parameter cannot be assigned a value</td>
<td>Formal parameter must be assigned a value</td>
<td>Formal parameter should be assigned a value</td>
<td></td>
</tr>
<tr>
<td>Actual parameter can be a constant, initialized variable, literal, or expression</td>
<td>Actual parameter must be a variable</td>
<td>Actual parameter must be a variable</td>
<td></td>
</tr>
<tr>
<td>Actual parameter is passed by reference (the caller passes the subprogram a pointer to the value)</td>
<td>Actual parameter is passed by value (the subprogram passes the caller a copy of the value) unless NOCOPY is specified</td>
<td>Actual parameter is passed by value (the caller passes the subprogram a copy of the value) unless NOCOPY is specified</td>
<td></td>
</tr>
</tbody>
</table>

**Specifying 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 Passing Actual Subprogram Parameters with Positional, Named, or Mixed Notation on page 8-11).

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 8–6 illustrates the use of default values for subprogram parameters.

**Example 8–6 Procedure with Default Parameter Values**

```plsql
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, as in Example 8–7.

**Example 8–7 Formal Parameter with Expression as Default Value**

```plsql
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 8–7 is:

```
1
2
```
Passing Actual Subprogram Parameters with Positional, Named, or Mixed Notation

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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positional</td>
<td>Specify the same parameters in the same order as the procedure declares them.</td>
<td>Compact and readable, but has these disadvantages:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ If you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ If the procedure’s parameter list changes, you must change your code.</td>
</tr>
<tr>
<td>Named</td>
<td>Specify the name and value of each parameter, using the association operator, =&gt;. Order of parameters is insignificant.</td>
<td>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.</td>
</tr>
<tr>
<td>Mixed</td>
<td>Start with positional notation, then use named notation for the remaining parameters.</td>
<td>Recommended when you invoke procedures that have required parameters followed by optional parameters, and you must specify only a few of the optional parameters.</td>
</tr>
</tbody>
</table>

Example 8–8 shows equivalent subprogram calls using positional, named, and mixed notation.

Example 8–8 Subprogram Calls Using Positional, Named, and Mixed Notation

```sql
SQL> DECLARE
2    emp_num NUMBER(6) := 120;
3    bonus   NUMBER(6) := 50;
4  PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
5      BEGIN
6        UPDATE employees SET salary =
7          salary + amount WHERE employee_id = emp_id;
8    END raise_salary;
9  BEGIN
10     -- Positional notation:
11    raise_salary(emp_num, bonus);
12     -- Named notation (parameter order is insignificant):
13    raise_salary(amount => bonus, emp_id => emp_num);
14    raise_salary(emp_id => emp_num, amount => bonus);
15     -- Mixed notation:
16    raise_salary(emp_num, amount => bonus);
17  END;
18 / 
```
Overloading PL/SQL Subprogram Names

PL/SQL procedure successfully completed.

SQL> REM Clean up
SQL> ROLLBACK;

Rollback complete.

SQL> CREATE OR REPLACE FUNCTION compute_bonus (emp_id NUMBER, bonus NUMBER)
2    RETURN NUMBER
3  IS
4    emp_sal NUMBER;
5  BEGIN
6    SELECT salary INTO emp_sal
7      FROM employees
8      WHERE employee_id = emp_id;
9    RETURN emp_sal + bonus;
10  END compute_bonus;
11  /

Function created.

SQL> SELECT compute_bonus(120, 50) FROM DUAL; -- positional
SQL> SELECT compute_bonus(bonus => 50, emp_id => 120) FROM DUAL; -- named
SQL> SELECT compute_bonus(120, bonus => 50) FROM DUAL; -- mixed

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 data type 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 data type. This technique is useful in writing mathematical application programming interfaces (APIs), because several versions of a function can use the same name, and each can accept a different numeric type. For example, a function that accepts BINARY_FLOAT might be faster, while a function that accepts BINARY_DOUBLE might provide more precision.

To avoid problems or unexpected results passing parameters to such overloaded subprograms:

- Ensure that the expected version of a subprogram is invoked for each set of expected parameters. For example, if you have overloaded functions that accept BINARY_FLOAT and BINARY_DOUBLE, which is invoked if you pass a VARCHAR2 literal such as '5.0'?

- Qualify numeric literals and use conversion functions to make clear what the intended parameter types are. For example, use literals such as 5.0f (for BINARY_FLOAT), 5.0d (for BINARY_DOUBLE), or conversion functions such as TO_BINARY_FLOAT, TO_BINARY_DOUBLE, and TO_NUMBER.

PL/SQL looks for matching numeric parameters in this order:

1. PLS_INTEGER (or BINARY_INTEGER, an identical data type)
2. NUMBER
3. BINARY_FLOAT
4. BINARY_DOUBLE

A VARCHAR2 value can match a NUMBER, BINARY_FLOAT, or BINARY_DOUBLE parameter.

PL/SQL uses the first overloaded subprogram that matches the supplied parameters. For example, the SQRT function takes a single parameter. There are overloaded versions that accept a NUMBER, a BINARY_FLOAT, or a BINARY_DOUBLE parameter. If you pass a PLS_INTEGER parameter, the first matching overload is the one with a NUMBER parameter.
The `SQRT` function that takes a `NUMBER` parameter is likely to be slowest. To use a faster version, use the `TO_BINARY_FLOAT` or `TO_BINARY_DOUBLE` function to convert the parameter to another data type before passing it to the `SQRT` function.

If PL/SQL must convert a parameter to another data type, it first tries to convert it to a higher data type. For example:

- The `ATAN2` function takes two parameters of the same type. If you pass parameters of different types—for example, one `PLS_INTEGER` and one `BINARY_FLOAT`—PL/SQL tries to find a match where both parameters use the higher type. In this case, that is the version of `ATAN2` that takes two `BINARY_FLOAT` parameters; the `PLS_INTEGER` parameter is converted upwards.

- A function takes two parameters of different types. One overloaded version takes a `PLS_INTEGER` and a `BINARY_FLOAT` parameter. Another overloaded version takes a `NUMBER` and a `BINARY_DOUBLE` parameter. If you invoke this function and pass two `NUMBER` parameters, PL/SQL first finds the overloaded version where the second parameter is `BINARY_FLOAT`. Because this parameter is a closer match than the `BINARY_DOUBLE` parameter in the other overload, PL/SQL then looks downward and converts the first `NUMBER` parameter to `PLS_INTEGER`.

### Restrictions on Overloading

You cannot overload the following subprograms:

- **Standalone subprograms**

- **Subprograms whose formal parameters differ only in mode; for example:**

  ```plsql
  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:**

  ```plsql
  PACKAGE pkg IS
      PROCEDURE s (p INTEGER);
      PROCEDURE s (p REAL);
  END pkg;
  ```

  `INTEGER` and `REAL` are subtypes of `NUMBER`, so they belong to the same data type family.

- **Functions that differ only in return value data type, even if the data types are in different families; for example:**

  ```plsql
  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

```plsql
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

```plsql
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

```plsql
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:

```plsql
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:

```plsql
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 data type (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**

```sql
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 exception');
        --      swap(num1, num2);
        --      wrong number or types of arguments in call to 'SWAP'
```
Using Invoker's Rights or Definer's Rights (AUTHID Clause)

The AUTHID property of a stored PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at run time. The AUTHID property does not affect compilation, and has no meaning for units that have no code, such as collection types.

AUTHID property values are exposed in the static data dictionary view *
PROCEDURES. For units for which AUTHID has meaning, the view shows the value CURRENT_USER or DEFINER; for other units, the view shows NULL.

For stored PL/SQL units that you create or alter with the following statements, you can use the optional AUTHID clause to specify either CURRENT_USER or DEFINER. The default is DEFINER.

- CREATE FUNCTION Statement on page 14-29
- CREATE PACKAGE Statement on page 14-39
- CREATE PROCEDURE Statement on page 14-46
- CREATE TYPE Statement on page 14-65
- ALTER TYPE Statement on page 14-15

A unit whose AUTHID value is CURRENT_USER is called an invoker's rights unit, or IR unit. A unit whose AUTHID value is DEFINER is called a definer's rights unit, or DR unit. An anonymous block always behaves like an IR unit. A trigger or view always behaves like a DR unit.

The AUTHID property of a unit determines whether the unit is IR or DR, and it affects both name resolution and privilege checking at run time:

- The context for name resolution is CURRENT_SCHEMA.
- The privileges checked are those of the CURRENT_USER and the enabled roles.

When a session starts, CURRENT_SCHEMA has the value of the schema owned by SESSION_USER, and CURRENT_USER has the same value as SESSION_USER. (To get the current value of CURRENT_SCHEMA, CURRENT_USER, or SESSION_USER, use the SYS_CONTEXT function, documented in Oracle Database SQL Language Reference.)

CURRENT_SCHEMA can be changed during the session with the SQL statement ALTER SESSION SET CURRENT_SCHEMA. CURRENT_USER cannot be changed programmatically, but it might change when a PL/SQL unit or a view is pushed onto, or popped from, the call stack.

**Note:** Oracle recommends against issuing ALTER SESSION SET CURRENT_SCHEMA from within a stored PL/SQL unit.

During a server call, when a DR unit is pushed onto the call stack, the database stores the currently enabled roles and the current values of CURRENT_USER and CURRENT_SCHEMA. It then changes both CURRENT_USER and CURRENT_SCHEMA to the owner of
the DR unit, and enables only the role **PUBLIC**. (The stored and new roles and values are not necessarily different.) When the DR unit is popped from the call stack, the database restores the stored roles and values. In contrast, when an IR unit is pushed onto, or popped from, the call stack, the values of **CURRENT_USER** and **CURRENT_SCHEMA**, and the currently enabled roles do not change.

For dynamic SQL statements issued by a PL/SQL unit, name resolution and privilege checking are done only once, at run time. For static SQL statements, name resolution and privilege checking are done twice: first, when the PL/SQL unit is compiled, and then again at run time. At compilation time, the **AUTHID** property has no effect—both DR and IR units are treated like DR units. At run time, however, the **AUTHID** property determines whether a unit is IR or DR, and the unit is treated accordingly.

Topics:
- Choosing Between **AUTHID CURRENT_USER** and **AUTHID DEFINER**
- **AUTHID** and the SQL Command **SET ROLE**
- Need for Template Objects in IR Subprograms
- Overriding Default Name Resolution in 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

Choosing Between **AUTHID CURRENT_USER** and **AUTHID DEFINER**

**Scenario:** Suppose that you want to create an API whose procedures have unrestricted access to its tables, but you want to prevent ordinary users from selecting table data directly, and from changing it with **INSERT**, **UPDATE**, and **DELETE** statements.

**Solution:** In a special schema, create the tables and the procedures that comprise the API. By default, each procedure is a DR unit, so you need not specify **AUTHID DEFINER** when you create it. To other users, grant the **EXECUTE** privilege, but do not grant any privileges that allow data access.

**Scenario:** Suppose that you want to write a PL/SQL procedure that presents compilation errors to a developer. The procedure will join the static data dictionary views **ALL_SOURCE** and **ALL_ERRORS** and use the procedure **DBMS_OUTPUT.PUT_LINE** to show a window of numbered source lines around each error, following the list of errors for that window. You want all developers to be able to execute the procedure, and you want the procedure to treat each developer as the **CURRENT_USER** with respect to **ALL_SOURCE** and **ALL_ERRORS**.

**Solution:** When you create the procedure, specify **AUTHID CURRENT_USER**. Grant the **EXECUTE** privilege to **PUBLIC**. Because the procedure is an IR unit, **ALL_SOURCE** and **ALL_ERRORS** will operate from the perspective of the user who invokes the procedure.

---

**Note:** Another solution is to make the procedure a DR unit and grant its owner the **SELECT** privilege on both **DBA_SOURCE** and **DBA_ERRORS**. However, this solution is harder to program, and far harder to audit with respect to the criterion that a user must never see source code for units for which he or she does not have the **EXECUTE** privilege.
AUTHID and the SQL Command SET ROLE

The SQL command SET ROLE succeeds only if there are no DR units on the call stack. If at least one DR unit is on the call stack, issuing the SET ROLE command causes ORA-06565.

---

**Note:** To execute the SET ROLE command from PL/SQL, you must use dynamic SQL, preferably the EXECUTE IMMEDIATE statement. For information about this statement, see Using the EXECUTE IMMEDIATE Statement on page 7-2.

---

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

Sometimes, the run-time name resolution rules for an IR unit (that cause different invocations to resolve the same unqualified name to different objects) are not desired. Rather, it is required that a specific object be used on every invocation. Nevertheless, an IR unit is needed for other reasons. For example, it might be critical that privileges are evaluated with respect to the CURRENT_USER. Under these circumstances, qualify the name with the schema that owns the object.

Notice that an unqualified name that intends to denote a public synonym is exposed to the risk of capture if the schema of the CURRENT_USER has a colliding name. A public synonym can be qualified with "PUBLIC". You must enclose PUBLIC in double quotation marks. For example:

```
SELECT sysdate INTO today FROM 'PUBLIC'.DUAL;
```

---

**Note:** Oracle recommends against issuing the SQL statement ALTER SESSION SET CURRENT_SCHEMA from within a stored PL/SQL unit.

---

Using Views and Database Triggers with IR Subprograms

For IR subprograms executed within a view expression, the user who created the view, not the user who is querying the view, is considered to be the current user. This rule also applies to database triggers.

---

**Note:** If SYS_CONTEXT is used directly in the defining SQL statement of a view, then the value it returns for CURRENT_USER is the querying user and not the owner of the view.

---

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, the database 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 about object types, see Oracle Database Object-Relational Developer’s Guide.

Suppose that user HR creates the object type in Example 8–16.

**Example 8–16  Creating an Object Type with AUTHID CURRENT USER**

```sql
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:

```
CREATE TABLE person_tab OF hr.person_typ;
```
BEGIN
    hr.person_typ.new_person_typ(1001,
        'Jane Smith',
        'CLERK',
        'oe',
        'person_tab');
    hr.person_typ.new_person_typ(1002,
        'Joe Perkins',
        'SALES', 'oe',
        'person_tab');
    hr.person_typ.new_person_typ(1003,
        'Robert Lange',
        'DEV',
        'oe', 'person_tab');
END;
/

The calls succeed because the procedure 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 8–16, 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 8–17.

**Example 8–17 Invoking an IR Instance Methods**

```sql
-- 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;

-- 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 database initialization parameter OPEN_CURSORS.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a terminating condition. Otherwise, the recursion continues until PL/SQL runs out of memory and raises the predefined exception STORAGE_ERROR.

Recursion is a powerful technique for simplifying the design of algorithms. Basically, recursion means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21,...), is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined as simpler versions of itself. Consider the definition of n factorial (n!), the product of all integers from 1 to n:

\[ n! = n \times (n - 1)! \]

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 8–18.

```java
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 8–18 and then can invoke the procedure `raise_salary` from an anonymous PL/SQL block.

**Example 8–18 Invoking an External Procedure from PL/SQL**

```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 8–19 shows a call to a Java function from a PL/SQL procedure.

**Example 8–19 Invoking a Java Function from PL/SQL**

```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
    -- CREATE 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_REFERENCESPragma` on page 13-107).

In Example 8–20, the `RESTRICT_REFERENCES` pragma asserts that packaged function `credit_ok` writes no database state (`WNDS`) and reads no package state (`RNPS`).

Example 8–20  `RESTRICT_REFERENCES` Pragma

```sql
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 8–21, 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 8–21  Aliasing from Passing Global Variable with `NOCOPY` Hint

```sql
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 8–22, 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 8–22  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 8–23, 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 8–23  Aliasing from Assigning Cursor Variables to Same Work Area

DECLARE
  TYPE EmpCurTyp IS REF CURSOR;
Using the PL/SQL Function Result Cache

The PL/SQL function result caching mechanism provides a language-supported and system-managed means for caching the results of PL/SQL functions in a shared global area (SGA), which is available to every session that runs your application. The caching mechanism is both efficient and easy to use, and 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 reexecute the function body. If the cache does not contain the result, the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the 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

```plsql
-- Using the PL/SQL Function Result Cache

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;
/```
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 `RELIERS_ON` clause, specify any tables or views on which the function results depend.

For the syntax of the `RESULT_CACHE` and `RELIERS_ON` clauses, see Function Declaration and Definition on page 13-75.

In Example 8–24, the package `department_pks` 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 8–24 Declaration and Definition of Result-Cached Function

```plsql
-- 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 RELIERS_ON (EMPLOYEES)
  IS
    rec dept_info_record;
    BEGIN
      SELECT AVG(SALARY), COUNT(*) INTO rec
        FROM EMPLOYEES
        WHERE DEPARTMENT_ID = dept_id;
      RETURN rec;
    END get_dept_info;
END department_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:

```sql
department_pks.get_dept_info(10);
```

The following call returns only the average salary in department number 10:

```sql
department_pks.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 `department_pks.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 8-33)

### 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:
Using the PL/SQL Function Result Cache

- 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 e-mail.
- It does not depend on session-specific settings.
  For more information, see Making Result-Cached Functions Handle Session-Specific Settings on page 8-33.
- It does not depend on session-specific application contexts.
  For more information, see Making Result-Cached Functions Handle Session-Specific Application Contexts on page 8-34.

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
Using the PL/SQL Function Result Cache

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 \texttt{get\_value}. Given a parameter name, application ID, and role ID, \texttt{get\_value} returns the setting that applies to the parameter.

The function \texttt{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 \texttt{global\_config\_params}, \texttt{app\_level\_config\_params}, or \texttt{role\_level\_config\_params} invalidates the cached results of \texttt{get\_value}, include their names in the \texttt{RELIES\_ON} clause.

Example 8–25 shows a possible definition for \texttt{get\_value}.

**Example 8–25 Result-Cached Function that Returns Configuration Parameter Setting**

```sql
CREATE OR REPLACE 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 `RELIRES_ON` clause is unnecessary.

```
CREATE OR REPLACE FUNCTION fibonacci (n NUMBER) RETURN NUMBER RESULT_CACHE IS
BEGIN
    IF (n = 0) OR (n = 1) THEN
        RETURN 1;
    ELSE
        RETURN fibonacci(n - 1) + fibonacci(n - 2);
    END IF;
END;
/
```

**Advanced Result-Cached Function Topics**

Topics:

- Rules for a Cache Hit
- 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 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:

<table>
<thead>
<tr>
<th>Cache Hit Rules</th>
<th>&quot;Equal To&quot; Operator Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL is the same as NULL</td>
<td>NULL = NULL evaluates to NULL.</td>
</tr>
</tbody>
</table>
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 Units on Which Result-Cached Functions Depend on page 8-38).
- 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 PL/SQL function result cache has only committed changes that are visible to all sessions, so that uncommitted changes in one session are not visible to other sessions.

Making Result-Cached Functions Handle Session-Specific Settings
If a function depends on settings that might vary from session to session (such as NLS_DATE_FORMAT and TIME_ZONE), make the function result-cached only if you can modify it to handle the various settings.

Consider the following function:

Example 8–26
CREATE OR REPLACE 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

<table>
<thead>
<tr>
<th>Cache Hit Rules</th>
<th>&quot;Equal To&quot; Operator Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 &quot;equal to&quot; operator.)</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:
  ```sql
  TO_CHAR(date_hired, 'mm/dd/yy');
  ```
- Add a format mask parameter to get_hire_date. For example:
  ```sql
  CREATE OR REPLACE 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 8–27, assume that a table, config_tab, has a VPD policy that translates this query:

```sql
SELECT value FROM config_tab
WHERE name = param_name;
```

to this query:

```sql
SELECT value FROM config_tab
WHERE name = param_name
  AND app_id = SYS_CONTEXT('Config', 'App_ID');
```
Example 8–27  Result-Cached Function that Depends on Session-Specific Application Context

CREATE OR REPLACE FUNCTION get_param_value
    (param_name VARCHAR,
     appctx VARCHAR DEFAULT SYS_CONTEXT('Config', 'App_ID'))
RETURN VARCHAR
RESULT_CACHE RELIES_ON (config_tab)
IS
    rec VARCHAR(2000);
BEGIN
    SELECT value INTO rec
    FROM config_tab
    WHERE Name = param_name;
END;
/

Choosing Result-Caching Granularity

PL/SQL provides the function result cache, but you choose the caching granularity. To understand the concept of granularity, consider the Product_Descriptions table in the Order Entry (OE) sample schema:

<table>
<thead>
<tr>
<th>NAME</th>
<th>NULL?</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCT_ID</td>
<td>NOT NULL</td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>LANGUAGE_ID</td>
<td>NOT NULL</td>
<td>VARCHAR2(3)</td>
</tr>
<tr>
<td>TRANSLATED_NAME</td>
<td>NOT NULL</td>
<td>NVARCHAR2(50)</td>
</tr>
<tr>
<td>TRANSLATED_DESCRIPTION</td>
<td>NOT NULL</td>
<td>NVARCHAR2(2000)</td>
</tr>
</tbody>
</table>

The table has the name and description of each product in several languages. The unique key for each row is PRODUCT_ID, LANGUAGE_ID.

Suppose that you want to define a function that takes a PRODUCT_ID and a LANGUAGE_ID and returns the associated TRANSLATED_NAME. You also want to cache the translated names. Some of the granularity choices for caching the names are:

- One name at a time (finer granularity)
- One language at a time (coarser granularity)

Table 8–3  Comparison of Finer and Coarser Caching Granularity

<table>
<thead>
<tr>
<th>Finer Granularity</th>
<th>Coarser Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each function result corresponds to one logical result.</td>
<td>Each function result contains many logical subresults.</td>
</tr>
<tr>
<td>Stores only data that is needed at least once.</td>
<td>Might store data that is never used.</td>
</tr>
<tr>
<td>Each data item ages out individually.</td>
<td>One aged-out data item ages out the whole set.</td>
</tr>
<tr>
<td>Does not allow bulk loading optimizations.</td>
<td>Allows bulk loading optimizations.</td>
</tr>
</tbody>
</table>

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 8–28, get_product_name_1 is a result-cached function. Whenever get_product_name_1 is invoked with a different PRODUCT_ID and LANGUAGE_ID, it caches the associated TRANSLATED_NAME. Each call to get_product_name_1 adds at most one TRANSLATED_NAME to the cache.
Using the PL/SQL Function Result Cache

Example 8–28  Caching One Name at a Time (Finer Granularity)

```plsql
CREATE OR REPLACE FUNCTION get_product_name_1 (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 8–29, `get_product_name_2` defines a result-cached function, `all_product_names`. Whenever `get_product_name_2` invokes `all_product_names` with a different `LANGUAGE_ID`, `all_product_names` caches every `TRANSLATED_NAME` associated with that `LANGUAGE_ID`. Each call to `all_product_names` adds every `TRANSLATED_NAME` of at most one `LANGUAGE_ID` to the cache.

Example 8–29  Caching Translated Names One Language at a Time (Coarser Granularity)

```plsql
CREATE OR REPLACE FUNCTION get_product_name_2 (prod_id NUMBER, lang_id VARCHAR2)
RETURN NVARCHAR2
IS
    TYPE product_names IS TABLE OF NVARCHAR2(50) INDEX BY PLS_INTEGER;
    FUNCTION all_product_names (lang_id NUMBER) RETURN product_names
    RESULT_CACHE 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;
        RETURN all_names;
    END;
BEGIN
    RETURN all_product_names(lang_id)(prod_id);
END;
```

Result Caches in Oracle RAC Environment

Cached results are stored in the system global area (SGA). In an Oracle RAC environment, each database instance 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 about these parameters, see *Oracle Database Reference* and *Oracle Database Performance Tuning Guide*.

  **See Also:**

  - *Oracle Database Reference* for more information about **RESULT_CACHE_MAX_SIZE**
  - *Oracle Database Reference* for more information about **RESULT_CACHE_MAX_RESULT**
  - *Oracle Database Performance Tuning Guide* for more information about Result Cache concepts

- **DBMS_RESULT_CACHE** package

  The **DBMS_RESULT_CACHE** package provides an interface to allow the DBA to administer that part of the shared pool that is used by the SQL result cache and the PL/SQL function result cache. For more information about this package, see *Oracle Database PL/SQL Packages and Types Reference*.

- Dynamic performance views:
  
  - **[G]V$RESULT_CACHE_STATISTICS**
  - **[G]V$RESULT_CACHE_MEMORY**
  - **[G]V$RESULT_CACHE_OBJECTS**
  - **[G]V$RESULT_CACHE_DEPENDENCY**


**Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend**

When you hot-patch a PL/SQL unit on which a result-cached function depends (directly or indirectly), the cached results associated with the result-cached function might not be automatically flushed in all cases.

For example, suppose that the result-cached function **P1.foo()** depends on the packaged subprogram **P2.bar()**. If a new version of the body of package **P2** is loaded, the cached results associated with **P1.foo()** are not automatically flushed.

Therefore, this is the recommended procedure for hot-patching a PL/SQL unit:
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.
A trigger is a named PL/SQL 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.

Topics:
- Trigger Types
- Trigger States
- Data Access for Triggers
Uses of Triggers

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). The 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 9-8.

A system trigger is defined on a schema or the database. A trigger defined on a schema fires for each event associated with the owner of the schema (the current user). A trigger defined on a database fires for each event associated with all users.

A simple trigger can fire at exactly one of 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 9-13.

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: CREATE TRIGGER Statement on page 14-51

Data Access for Triggers

When a trigger is fired, the tables referenced in the trigger action might be currently undergoing changes by SQL statements in other users’ transactions. In all cases, the SQL statements running within triggers follow the common rules used for standalone SQL statements. In particular, if an uncommitted transaction has modified values that a trigger being fired either must read (query) or write (update), then the SQL statements in the body of the trigger being fired use the following guidelines:

- Queries see the current read-consistent materialized view of referenced tables and any data changed within the same transaction.
- Updates wait for existing data locks to be released before proceeding.
Uses of Triggers

Triggers supplement the standard capabilities of your database to provide a highly customized database management system. For example, you can use triggers to:

- Automatically generate derived column values
- Enforce referential integrity across nodes in a distributed database
- Enforce complex business rules
- Provide transparent event logging
- Provide auditing
- Maintain synchronous table replicates
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Publish information about database events, user events, and SQL statements to subscribing applications
- Restrict DML operations against a table to those issued during regular business hours
- Enforce security authorizations
- Prevent invalid transactions

Caution: Triggers are not reliable security mechanisms, because they are programmatic and easy to disable. For high assurance security, use Oracle Database Vault. For more information, see Oracle Database Vault Administrator’s Guide.

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 database features.

For example, do not define triggers to reject bad data if you can do the same checking through constraints.

Although you can use both triggers and integrity constraints to define and enforce any type of integrity rule, Oracle strongly recommends that you use triggers to constrain data input only in the following situations:

- To enforce referential integrity when child and parent tables are on different nodes of a distributed database
- To enforce complex business rules not definable using integrity constraints
- When a required referential integrity rule cannot be enforced using the following integrity constraints:
  - NOT NULL
  - PRIMARY KEY
  - FOREIGN KEY
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

Limit the size of triggers.

If the logic for your trigger requires much more than 60 lines of PL/SQL code, put most of the code in a stored subprogram and invoke the subprogram from the trigger.

The size of the trigger cannot exceed 32K.

Use triggers only for centralized, global operations that must fire for the triggering statement, regardless of which user or database application issues the statement.

Do not create recursive triggers.

For example, if you create an AFTER UPDATE statement trigger on the employees table, and the trigger itself issues an UPDATE statement on the employees table, the trigger fires recursively until it runs out of memory.

Use triggers on DATABASE judiciously. They are executed for every user every time the event occurs on which the trigger is created.

If you use a LOGON trigger to monitor logons by users, include an exception-handling part in the trigger, and include a WHEN OTHERS exception in the exception-handling part. Otherwise, an unhandled exception might block all connections to the database.

If you use a LOGON trigger only to execute a package (for example, an application context-setting package), put the exception-handling part in the package instead of in the trigger.
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;
/
```

The trigger in Example 9–1 fires when DML operations are performed on the table. You can choose what combination of operations must fire the trigger.

Because the trigger uses the BEFORE keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error by assigning to :NEW.column_name. You might use the AFTER keyword if you want the trigger to query or change the same table, because triggers can only do that after the initial changes are applied and the table is back in a consistent state.

Because the trigger uses the FOR EACH ROW clause, it might 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 9-5 to show how parts of a trigger are specified. For additional examples of CREATE TRIGGER statements, see Examples of Trigger Applications on page 9-31.

Topics:
Creating Triggers

- 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 need not 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, database event, or DDL event that fires the trigger body. The options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, DATABASE, or SCHEMA on which the trigger is defined.

Note: Exactly one table or view can be specified in the triggering statement. If the INSTEAD OF option is used, then the triggering statement must specify a view; conversely, if a view is specified in the triggering statement, then only the INSTEAD OF option can be used.

In Example 9–1 on page 9-5, the PRINT_SALARY_CHANGES trigger fires after any DELETE, INSERT, or UPDATE on the 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:

```sql
... 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 9-13.

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.

**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 the database performs a transparent `ROLLBACK` to `SAVEPOINT` and restarts the update. This can occur many times before the statement completes successfully. Each
time the statement is restarted, the BEFORE statement trigger fires again. The rollback
to savepoint does not undo changes to any package variables referenced in the trigger.
Include a counter variable in your package to detect this situation.

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. The database allows up to 32 triggers to cascade at simultaneously.
You can limit the number of trigger cascades by using the initialization parameter
OPEN_CURSORS, because a cursor must be opened for every execution of a trigger.

Although any trigger can run a sequence of operations either inline or by invoking
subprograms, using multiple triggers of the same type allows the modular installation
of applications that have triggers on the same tables.

Each subsequent trigger sees the changes made by the previously fired triggers. Each
trigger can see the old and new values. The old values are the original values, and the
new values are the current values, as set by the most recently fired UPDATE or INSERT
trigger.

The 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, the database chooses an arbitrary, unpredictable order.

See Also: CREATE TRIGGER Statement on page 14-51 for more
information about ordering of triggers and the FOLLOWS clause

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 9-9.

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, the 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 9-12

---

**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.

```sql
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 Amp_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:

```sql
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:

```sql
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  
    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 9-13.

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, assume that the table Emp_log was created as follows:

CREATE TABLE Emp_log {
    Emp_id     NUMBER,
    Log_date   DATE,
    New_salary NUMBER,
    Action     VARCHAR2(20);}

Then, define the following trigger:

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 an optional declarative part 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**

SQL> CREATE OR REPLACE TRIGGER compound_trigger
2    FOR UPDATE OF salary ON employees
3      COMPOUND TRIGGER
4
5    -- Declarative part (optional)
6    -- Variables declared here have firing-statement duration.
7    threshold CONSTANT SIMPLE_INTEGER := 200;
8
9    BEFORE STATEMENT IS
10    BEGIN
11      NULL;
12    END BEFORE STATEMENT;
13
14    BEFORE EACH ROW IS
15    BEGIN
16      NULL;
17    END BEFORE EACH ROW;
18
19    AFTER EACH ROW IS
20    BEGIN
21      NULL;
22    END AFTER EACH ROW;
23
24    AFTER STATEMENT IS
25    BEGIN
26      NULL;
27    END AFTER STATEMENT;
28  END compound_trigger;
29  /

Trigger created.

SQL>

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 9-16)
- To avoid the mutating-table error (ORA-04091) (as in Using Compound Triggers to Avoid Mutating-Table Error on page 9-18)

**Compound Trigger Sections**

A compound trigger has a declarative part and at least one timing-point section. It cannot have multiple sections for the same timing point.

The optional declarative part (the first part) declares variables and subprograms that timing-point sections can use. When the trigger fires, the declarative part 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>
<thead>
<tr>
<th>Timing Point</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the triggering statement executes</td>
<td>BEFORE STATEMENT</td>
</tr>
<tr>
<td>After the triggering statement executes</td>
<td>AFTER STATEMENT</td>
</tr>
<tr>
<td>Before each row that the triggering statement affects</td>
<td>BEFORE EACH ROW</td>
</tr>
<tr>
<td>After each row that the triggering statement affects</td>
<td>AFTER EACH ROW</td>
</tr>
</tbody>
</table>

Any section can include the functions Inserting, Updating, Deleting, and Applying.

See Also: CREATE TRIGGER Statement on page 14-51 for more information about the syntax of compound triggers

Triggering Statements of Compound Triggers

The triggering statement of a compound trigger must be a DML statement.

If the triggering statement affects no rows, and the compound trigger has neither a BEFORE STATEMENT section nor an AFTER STATEMENT section, the trigger never fires.

It is when the triggering statement affects many rows that a compound trigger has a performance benefit. This is why it is important to use the BULK COLLECT clause with the FORALL statement. For example, without the BULK COLLECT clause, a FORALL statement that contains an INSERT statement simply performs a single-row insertion operation many times, and you get no benefit from using a compound trigger. For more information about using the BULK COLLECT clause with the FORALL statement, see Using FORALL and BULK COLLECT Together on page 12-21.

If the triggering statement of a compound trigger is an INSERT statement that includes a subquery, the compound trigger retains some of its performance benefit. For example, suppose that a compound trigger is triggered by the following statement:

```sql
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
  WHERE Source.c1 > 0
```

For each row of Source whose column c1 is greater than zero, the BEFORE EACH ROW and AFTER EACH ROW sections of the compound trigger execute. However, the 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 declarative part 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 declarative part, the BEFORE STATEMENT section, or the AFTER STATEMENT section.
Only the BEFORE EACH ROW section can change the value of :NEW.
If, after the compound trigger fires, 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**

```sql
CREATE TABLE employee_salaries 
( 
    employee_id NUMBER NOT NULL, 
    change_date DATE NOT NULL, 
    salary NUMBER(8,2) NOT NULL, 
    CONSTRAINT pk_employee_salaries PRIMARY KEY (employee_id, change_date), 
    CONSTRAINT fk_employee_salaries FOREIGN KEY (employee_id) 
        REFERENCES employees (employee_id) 
        ON DELETE CASCADE) /

CREATE OR REPLACE TRIGGER maintain_employee_salaries 
FOR UPDATE OF salary ON employees 
    COMPOUND TRIGGER
```
-- Declarative Part:
-- Choose small 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 9-25.

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

```sql
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 9-13).

The trigger body is either a CALL subprogram (a PL/SQL subprogram, or a Java subprogram encapsulated in a PL/SQL wrapper) or a PL/SQL block, and as such, it
can include SQL and PL/SQL statements. These statements are executed if the triggering statement is entered and if the trigger restriction (if any) evaluates to TRUE.

If the trigger body for a row trigger is a PL/SQL block (not a CALL subprogram), it can include the following constructs:

- REFERENCING clause, which can specify correlation names OLD, NEW, and PARENT
- Conditional predicates INSERTING, DELETING, and UPDATING

See Also: CREATE TRIGGER Statement on page 14-51 for syntax and semantics of this statement

The LOGON trigger in Example 9–5 executes the procedure sec_mgr.check_user after a user logs onto the database. The body of the trigger includes an exception-handling part, which includes a WHEN OTHERS exception that invokes RAISE_APPLICATION_ERROR.

Example 9–5  Monitoring Logons with a Trigger

```sql
CREATE OR REPLACE TRIGGER check_user
AFTER LOGON ON DATABASE
BEGIN
    sec_mgr.check_user;
EXCEPTION
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR
        (-20000, 'Unexpected error: '|| DBMS_Utility.Format_Error_Stack);
END;
/
```

Although triggers are declared using PL/SQL, they can call subprograms in other languages. The trigger in Example 9–6 invokes a Java subprogram.

Example 9–6 Invoking a Java Subprogram from a Trigger

```sql
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:

```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 (+旧_id.intValue() +", '"+ old_ename.toString() + ", BEFORE DELETE');
        stmt.executeUpdate (sql);
```
Coding the Trigger Body

```java
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, you couldn't change the new value for a LOB.
  -- Now, you 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 that the table new was created as follows:

CREATE TABLE new (
  field1        NUMBER,
  field2        VARCHAR2(20));

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:

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:

```sql
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:

```sql
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 (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, as of 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 data type 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:

```sql
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:

```sql
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON emp
FOR EACH ROW
```
BEGIN
  WHEN dblink is inaccessible, compilation fails here:
  INSERT INTO emp0Remote 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 the 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 emp0Remote 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 Data Types
- 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 COMPILe. 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 Data Types

LONG and LONG RAW data types 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 data type.
- If data from a LONG or LONG RAW column can be converted to a constrained data type (such as CHAR and VARCHAR2), then a LONG or LONG RAW column can be referenced in a SQL statement within a trigger. The maximum length for these data types is 32000 bytes.
- Variables cannot be declared using the LONG or LONG RAW data types.
- :NEW and :PARENT cannot be used with LONG or LONG RAW columns.

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 9-18.)

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 must 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 9-25 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 reuse existing primary key values, or track updates to foreign key values and modify the trigger to ensure that no row is updated twice.

That is the only problem with this technique for foreign key updates. The trigger cannot miss rows that were changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is invoked.

**System Trigger Restrictions**

Depending on the event, different event attribute functions are available. For example, certain DDL operations might not be allowed on DDL events. Check Event Attribute Functions on page 9-46 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:

```sql
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`. The 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:

```sql
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 COMPILER;
```

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 must perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

To disable a trigger, use the `ALTER TRIGGER` statement with the `DISABLE` option. For example, to disable the trigger named `Reorder`, enter 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 information about *_TRIGGERS static data dictionary views

For example, assume the following statement was used to create the Reorder trigger:

```sql
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:

```sql
SELECT Trigger_type, Triggering_event, Table_name
FROM USER_TRIGGERS
WHERE Trigger_name = 'REORDER';
```

<table>
<thead>
<tr>
<th>TYPE</th>
<th>TRIGGERING_STATEMENT</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFTER EACH ROW</td>
<td>UPDATE</td>
<td>INVENTORY</td>
</tr>
</tbody>
</table>

```sql
SELECT Trigger_body
FROM USER_TRIGGERS
WHERE Trigger_name = 'REORDER';
```

<table>
<thead>
<tr>
<th>TRIGGER_BODY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE</td>
<td>x NUMBER;</td>
</tr>
<tr>
<td>BEGIN</td>
<td></td>
</tr>
<tr>
<td>SELECT COUNT(*)</td>
<td>INTO x</td>
</tr>
<tr>
<td>FROM Pending_orders</td>
<td></td>
</tr>
<tr>
<td>WHERE Part_no =</td>
<td>:NEW.Part_no;</td>
</tr>
<tr>
<td>IF x = 0 THEN</td>
<td>INSERT INTO Pending_orders</td>
</tr>
<tr>
<td>END IF;</td>
<td>VALUES (:NEW.Part_no, :NEW.Reorder_quantity, sysdate);</td>
</tr>
<tr>
<td>END;</td>
<td></td>
</tr>
</tbody>
</table>
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 the database. For example, triggers are commonly used to:

- Provide sophisticated auditing
- Prevent invalid transactions
- Enforce referential integrity (either those actions not supported by declarative constraints or across nodes in a distributed database)
- Enforce complex business rules
- Enforce complex security authorizations
- Provide transparent event logging
- Automatically generate derived column values
- Enable building complex views that are updatable
- Track database events

This section provides an example of each of these trigger applications. These examples are not meant to be used exactly as written: They are provided to assist you in designing your own triggers.

Auditing with Triggers

Triggers are commonly used to supplement the built-in auditing features of the database. Although triggers can be written to record information similar to that recorded by the AUDIT statement, use triggers only when more detailed audit information is required. For example, use triggers to provide value-based auditing for each row.

Sometimes, the AUDIT statement is considered a security audit facility, while triggers can provide financial audit facility.

When deciding whether to create a trigger to audit database activity, consider what the database's auditing features provide, compared to auditing defined by triggers, as shown in Table 9–2.

<table>
<thead>
<tr>
<th>Audit Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DML and DDL Auditing</td>
<td>Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, triggers permit auditing of DML statements entered against tables, and DDL auditing at SCHEMA or DATABASE level.</td>
</tr>
<tr>
<td>Centralized Audit Trail</td>
<td>All database audit information is recorded centrally and automatically using the auditing features of the database.</td>
</tr>
</tbody>
</table>
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.

---

**Table 9–2 (Cont.) Comparison of Built-in Auditing and Trigger-Based Auditing**

<table>
<thead>
<tr>
<th>Audit Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarative Method</td>
<td>Auditing features enabled using the standard database features are easier to declare and maintain, and less prone to errors, when compared to auditing functions defined by triggers.</td>
</tr>
<tr>
<td>Auditing Options can be Audited</td>
<td>Any changes to existing auditing options can also be audited to guard against malicious database activity.</td>
</tr>
<tr>
<td>Session and Execution time Auditing</td>
<td>Using the database auditing features, records can be generated once every time an audited statement is entered (<strong>BY ACCESS</strong>) or once for every session that enters an audited statement (<strong>BY SESSION</strong>). Triggers cannot audit by session; an audit record is generated each time a trigger-audited table is referenced.</td>
</tr>
<tr>
<td>Auditing of Unsuccessful Data Access</td>
<td>Database auditing can be set to audit when unsuccessful data access occurs. However, unless autonomous transactions are used, any audit information generated by a trigger is rolled back if the triggering statement is rolled back. For more information about autonomous transactions, see <em>Oracle Database Concepts</em>.</td>
</tr>
<tr>
<td>Sessions can be Audited</td>
<td>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.</td>
</tr>
</tbody>
</table>
**Examples of Trigger Applications**

**Using Triggers**

```sql
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. */
```

**Note:** You might need to set up the following data structures for the examples to work:

```sql
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);
```
Examples of Trigger Applications

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,  
      $ /* ADD MORE COLUMNS */$  
    );  
  END IF;  
END;
Examples of Trigger Applications

Using Triggers

```sql
-- 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;
```

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 declarative constraint features, use triggers only to enforce complex business rules that cannot be defined using standard constraints. The declarative constraint features provided with the database offer 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:

- \texttt{UPDATE SET NULL}, and \texttt{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 \texttt{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 \texttt{PRIMARY} (or \texttt{UNIQUE} \texttt{KEY}) constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, then you can also declare the foreign key in the child table, but disable it. Disabling the trigger in the child table prevents the corresponding \texttt{PRIMARY KEY} constraint from being dropped (unless the \texttt{PRIMARY KEY} constraint is explicitly dropped with the \texttt{CASCADE} option).

To maintain referential integrity using triggers:

- For the child table, define a trigger that ensures that values inserted or updated in the foreign key correspond to values in the parent key.
- For the parent table, define one or more triggers that ensure the desired referential action (\texttt{RESTRICT}, \texttt{CASCADE}, or \texttt{SET NULL}) for values in the foreign key when values in the parent key are updated or deleted. No action is required for inserts into the parent table (no dependent foreign keys exist).

The following topics provide examples of the triggers necessary to enforce referential integrity:

- Foreign Key Trigger for Child Table
- \texttt{UPDATE and DELETE RESTRICT} Trigger for Parent Table
- \texttt{UPDATE and DELETE SET NULL} Triggers for Parent Table
- \texttt{DELETE Cascade} Trigger for Parent Table
- \texttt{UPDATE Cascade} Trigger for Parent Table
- Trigger for Complex Check Constraints
- Complex Security Authorizations and Triggers
- Transparent Event Logging and Triggers
- Derived Column Values and Triggers
- Building Complex Updatable Views Using Triggers
- Fine-Grained Access Control Using Triggers

The examples in the following sections use the \texttt{emp} and \texttt{dept} table relationship. Several of the triggers include statements that lock rows (\texttt{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 UPDATECASCADE 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);
C
```

-- 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'
  |l ' 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.Updateseq;
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:

```sql
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

Examples of Trigger Applications

Using Triggers

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;

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
  RAISE application_error (-20300,  
  'Salary ||TO_CHAR(:NEW.Sal)||' out of range for '  
  ||'job classification '||:NEW.Job  
  ||' for employee '||:NEW.Ename);
END IF;

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 the database. For example, a trigger can prohibit updates to salary data of the emp table during weekends, holidays, and nonworking hours.

When using a trigger to enforce a complex security authorization, it is best to use a BEFORE statement trigger. Using a BEFORE statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to run, so that no wasted work is done by an unauthorized statement.
- The security check is performed only once for the triggering statement, not for each row affected by the triggering statement.

This example shows a trigger used to enforce security.

This example shows a trigger used to enforce security.
Examples of Trigger Applications

```
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 '
      ||'emnployee 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.
Examples of Trigger Applications

**Note:** You might need to set up the following data structures for the example to work:

```sql
ALTER TABLE Emp99 ADD(
    Uppername VARCHAR2(20),
    Soundexname VARCHAR2(20));
```

**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>
<thead>
<tr>
<th>Booknum</th>
<th>Section</th>
<th>Title</th>
<th>Author</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>121001</td>
<td>Classic</td>
<td>Iliad</td>
<td>Homer</td>
<td>Y</td>
</tr>
<tr>
<td>121002</td>
<td>Novel</td>
<td>Gone with the Wind</td>
<td>Mitchell M</td>
<td>N</td>
</tr>
</tbody>
</table>

Library consists of `library_table(section)`.

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geography</td>
</tr>
<tr>
<td>Classic</td>
</tr>
</tbody>
</table>
Examples of Trigger Applications

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**

You can use LOGON triggers to execute the package associated with an application context. An application context captures session-related information about the user who is logging in to the database. From there, your application can control how much access this user has, based on his or her session information.

---

**Note:** If you have very specific logon requirements, such as preventing users from logging in from outside the firewall or after work hours, consider using Oracle Database Vault instead of LOGON triggers. With Oracle Database Vault, you can create custom rules to strictly control user access.

---

**See Also:**

- *Oracle Database Security Guide* for information about creating a LOGON trigger to run a database session application context package
- *Oracle Database Vault Administrator’s Guide* for information about Oracle Database Vault
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 Queueing engine. Publish/subscribe applications use the `DBMS_AQ.ENQUEUE` procedure, and other applications such as cartridges use callouts.

**See Also:**
- `ALTER TRIGGER Statement` on page 14-12
- *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 about event-specific attributes

For each supported database event, you can identify and predefine event-specific attributes for the event. You can choose the parameter list to be any of these attributes, along with other simple expressions. For callouts, these are passed as IN arguments.

**Error Handling**

Return status from publication callout functions for all events are ignored. For example, with SHUTDOWN events, the database cannot do anything with the return status.

**Execution Model**

Traditionally, triggers 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:

- 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. Oracle recommends you use these public synonyms whose names begin with `ora_`.
- `ora_name_list_t` is defined in package `DBMS_STANDARD` as:

  ```
  TYPE ora_name_list_t IS TABLE OF VARCHAR2(64);
  ```

### Table 9–3  System-Defined Event Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ora_client_ip_address</code></td>
<td>VARCHAR2</td>
<td>Returns IP address of the client in a LOGON event when the underlying protocol is TCP/IP</td>
<td>DECLARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v_addr VARCHAR2(11); BEGIN IF (ora_sysevent = 'LOGON') THEN v_addr := ora_client_ip_address; END IF END;</td>
<td></td>
</tr>
<tr>
<td><code>ora_database_name</code></td>
<td>VARCHAR2(50)</td>
<td>Database name.</td>
<td>DECLARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v_db_name VARCHAR2(50); BEGIN v_db_name := ora_database_name; END;</td>
<td></td>
</tr>
<tr>
<td><code>ora_des_encrypted_password</code></td>
<td>VARCHAR2</td>
<td>The DES-encrypted password of the user being created or altered.</td>
<td>IF (ora_dict_obj_type = 'USER') THEN INSERT INTO event_table VALUES (ora_des_encrypted_password); END IF</td>
</tr>
<tr>
<td><code>ora_dict_obj_name</code></td>
<td>VARCHAR(30)</td>
<td>Name of the dictionary object on which the DDL operation occurred.</td>
<td>INSERT INTO event_table VALUES ('Changed object is '</td>
</tr>
<tr>
<td><code>ora_dict_obj_name_list</code></td>
<td>PLS_INTEGER</td>
<td>Return the list of object names of objects being modified in the event.</td>
<td>DECLARE</td>
</tr>
<tr>
<td><code>(name_list OUT ora_name_list_t)</code></td>
<td></td>
<td>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;</td>
<td></td>
</tr>
<tr>
<td><code>ora_dict_obj_owner</code></td>
<td>VARCHAR(30)</td>
<td>Owner of the dictionary object on which the DDL operation occurred.</td>
<td>INSERT INTO event_table VALUES ('object owner is '</td>
</tr>
<tr>
<td><code>ora_dict_obj_owner_list</code></td>
<td>PLS_INTEGER</td>
<td>Returns the list of object owners of objects being modified in the event.</td>
<td>DECLARE</td>
</tr>
<tr>
<td><code>(owner_list OUT ora_name_list_t)</code></td>
<td></td>
<td>owner_list DBMS_STANDARD.ora_name_list_t; number_modified PLS_INTEGER; BEGIN IF (ora_sysevent='ASSOCIATE STATISTICS') THEN number_modified := ora_dict_obj_owner_list(owner_list); END IF END;</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>ora_dict_obj_type</td>
<td>VARCHAR(20)</td>
<td>Type of the dictionary object on which the DDL operation occurred.</td>
<td>INSERT INTO event_table VALUES ('This object is a '</td>
</tr>
<tr>
<td>ora_grantee</td>
<td>PLS_INTEGER</td>
<td>Returns the grantees of a grant event in the OUT parameter; returns the number of grantees in the return value.</td>
<td>DECLARE 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;</td>
</tr>
<tr>
<td>ora_instance_num</td>
<td>NUMBER</td>
<td>Instance number.</td>
<td>IF (ora_instance_num = 1) THEN INSERT INTO event_table VALUES ('1'); END IF;</td>
</tr>
<tr>
<td>ora_is_alter_column</td>
<td>BOOLEAN</td>
<td>Returns true if the specified column is altered.</td>
<td>IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN alter_column := ora_is_alter_column('C'); END IF;</td>
</tr>
<tr>
<td>ora_is_creating_nested_table</td>
<td>BOOLEAN</td>
<td>Returns true if the current event is creating a nested table.</td>
<td>IF (ora_sysevent = 'CREATE' AND ora_dict_obj_type = 'TABLE' AND ora_is_creating_nested_table) THEN INSERT INTO event_table VALUES ('A nested table is created'); END IF;</td>
</tr>
<tr>
<td>ora_is_drop_column</td>
<td>BOOLEAN</td>
<td>Returns true if the specified column is dropped.</td>
<td>IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN drop_column := ora_is_drop_column('C'); END IF;</td>
</tr>
<tr>
<td>ora_is_servererror</td>
<td>BOOLEAN</td>
<td>Returns TRUE if given error is on error stack, FALSE otherwise.</td>
<td>IF ora_is_servererror(error_number) THEN INSERT INTO event_table VALUES ('Server error!!'); END IF;</td>
</tr>
<tr>
<td>ora_login_user</td>
<td>VARCHAR2(30)</td>
<td>Login user name.</td>
<td>SELECT ora_login_user FROM DUAL;</td>
</tr>
<tr>
<td>ora_partition_pos</td>
<td>PLS_INTEGER</td>
<td>In an INSTEAD OF trigger for CREATE TABLE, the position within the SQL text where you can insert a PARTITION clause.</td>
<td>-- Retrieve ora_sql_txt into -- sql_text variable first. v_n := ora_partition_pos; v_new_stmt := SUBSTR(sql_text,1,v_n - 1)</td>
</tr>
<tr>
<td>ora_privilege_list</td>
<td>PLS_INTEGER</td>
<td>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.</td>
<td>DECLARE privilege_list DBMS_STANDARD.ora_name_list_t; number_of_privileges PLS_INTEGER; BEGIN IF (ora_sysevent = 'GRANT' OR ora_sysevent = 'REVOKE') THEN number_of_privileges := ora_privilege_list(privilege_list); END IF; END;</td>
</tr>
</tbody>
</table>
table 9–3 (cont.) system-defined event attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ora_revokee</td>
<td>PLS_INTEGER</td>
<td>Returns the revokees of a revoke event in the OUT parameter; returns the number of revokees in the return value.</td>
<td>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;</td>
</tr>
<tr>
<td>ora_server_error</td>
<td>NUMBER</td>
<td>Given a position (1 for top of stack), it returns the error number at that position on error stack.</td>
<td>INSERT INTO event_table VALUES ('top stack error'</td>
</tr>
<tr>
<td>ora_server_error_depth</td>
<td>PLS_INTEGER</td>
<td>Returns the total number of error messages on the error stack.</td>
<td>n := ora_server_error_depth; -- This value is used with other functions such as ora_server_error</td>
</tr>
<tr>
<td>ora_server_error_msg</td>
<td>VARCHAR2</td>
<td>Given a position (1 for top of stack), it returns the error message at that position on error stack.</td>
<td>INSERT INTO event_table VALUES ('top stack error message'</td>
</tr>
<tr>
<td>ora_server_error_num_params</td>
<td>PLS_INTEGER</td>
<td>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.</td>
<td>n := ora_server_error_num_params(1);</td>
</tr>
<tr>
<td>ora_server_error_param</td>
<td>VARCHAR2</td>
<td>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.</td>
<td>-- For example, the second %s in a message: &quot;Expected %s, found %s&quot; param := ora_server_error_param(1,2);</td>
</tr>
<tr>
<td>ora_sql_txt</td>
<td>PLS_INTEGER</td>
<td>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.</td>
<td>--... -- 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</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Event</th>
<th>When Trigger Fires</th>
<th>Conditions</th>
<th>Restrictions</th>
<th>Transaction</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>STARTUP</td>
<td>When the database is opened.</td>
<td>None allowed</td>
<td>No database operations allowed in the trigger. Return status ignored.</td>
<td>Starts a separate transaction and commits it after firing the triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>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.</td>
<td>None allowed</td>
<td>No database operations allowed in the trigger. Return status ignored.</td>
<td>Starts a separate transaction and commits it after firing the triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name</td>
</tr>
<tr>
<td>DB_ROLE_CHANGE</td>
<td>When the database is opened for the first time after a role change.</td>
<td>None allowed</td>
<td>Return status ignored.</td>
<td>Starts a separate transaction and commits it after firing the triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name</td>
</tr>
<tr>
<td>SERVERERROR</td>
<td>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.</td>
<td>ERRNO = eno</td>
<td>Depends on the error. Return status ignored.</td>
<td>Starts a separate transaction and commits it after firing the triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name, ora_server_error, ora_is_servererror, space_error_info</td>
</tr>
</tbody>
</table>
Client Events

Client events are the events related to user logon/logoff, DML, and DDL operations.

The LOGON and LOGOFF events allow simple conditions on UID and USER. All other events allow simple conditions on the type and name of the object, 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>
<thead>
<tr>
<th>Table 9–5  Client Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
</tr>
</tbody>
</table>
| BEFORE ALTER            | When a catalog object is altered. | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_type  
or_dict_obj_name  
or_dict_obj_owner  
or_des_encrypted_password  
(for ALTER USER events)  
or_is_alter_column  
(for ALTER TABLE events)  
or_is_drop_column  
(for ALTER TABLE events) |
| AFTER ALTER             | When a catalog object is altered. | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_type  
or_dict_obj_name  
or_dict_obj_owner |
| BEFORE DROP             | When a catalog object is dropped. | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_type  
or_dict_obj_name  
or_dict_obj_owner |
| AFTER DROP              | When a catalog object is dropped. | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_type  
or_dict_obj_name  
or_dict_obj_owner |
| BEFORE ANALYZE          | When an analyze statement is issued | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_name  
or_dict_obj_type  
or_dict_obj_owner |
| AFTER ANALYZE           | When an analyze statement is issued | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_name  
or_dict_obj_type  
or_dict_obj_owner |
| BEFORE ASSOCIATE STATISTICS | When an associate statistics statement is issued | ora_sysevent  
or_login_user  
or_instance_num  
or_database_name  
or_dict_obj_name  
or_dict_obj_type  
or_dict_obj_owner  
or_dict_obj_name_list  
or_dict_obj_owner_list |
### Table 9-5 (Cont.) Client Events

<table>
<thead>
<tr>
<th>Event</th>
<th>When Trigger Fires</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE AUDIT</td>
<td>When an audit or noaudit statement is issued</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong></td>
</tr>
<tr>
<td>AFTER AUDIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE NOAUDIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER NOAUDIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE COMMENT</td>
<td>When an object is commented</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong> <strong>ora_dict_obj_name</strong> <strong>ora_dict_obj_type</strong> <strong>ora_dict_obj_owner</strong></td>
</tr>
<tr>
<td>AFTER COMMENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE CREATE</td>
<td>When a catalog object is created</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong> <strong>ora_dict_obj_type</strong> <strong>ora_dict_obj_name</strong> <strong>ora_dict_obj_owner</strong> <strong>ora_is_creating_nested_table</strong> (for CREATE TABLE events)</td>
</tr>
<tr>
<td>AFTER CREATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE DDL</td>
<td>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.</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong> <strong>ora_dict_obj_name</strong> <strong>ora_dict_obj_type</strong> <strong>ora_dict_obj_owner</strong></td>
</tr>
<tr>
<td>AFTER DDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE DISASSOCIATE STATISTICS</td>
<td>When a disassociate statistics statement is issued</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong> <strong>ora_dict_obj_name</strong> <strong>ora_dict_obj_type</strong> <strong>ora_dict_obj_owner</strong> <strong>ora_dict_obj_name_list</strong> <strong>ora_dict_obj_owner_list</strong></td>
</tr>
<tr>
<td>AFTER DISASSOCIATE STATISTICS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE GRANT</td>
<td>When a grant statement is issued</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong> <strong>ora_dict_obj_name</strong> <strong>ora_dict_obj_type</strong> <strong>ora_dict_obj_owner</strong> <strong>ora_grantee</strong> <strong>ora_with_grant_option</strong> <strong>ora_privileges</strong></td>
</tr>
<tr>
<td>AFTER GRANT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE LOGOFF</td>
<td>At the start of a user logoff</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong></td>
</tr>
<tr>
<td>AFTER LOGON</td>
<td>After a successful logon of a user.</td>
<td><strong>ora_sysevent</strong> <strong>ora_login_user</strong> <strong>ora_instance_num</strong> <strong>ora_database_name</strong> <strong>ora_client_ip_address</strong></td>
</tr>
</tbody>
</table>
### Table 9–5  (Cont.)  Client Events

<table>
<thead>
<tr>
<th>Event</th>
<th>When Trigger Fires</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE RENAME</td>
<td>When a rename statement is issued.</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_dict_obj_name</code>&lt;br&gt;<code>ora_dict_obj_owner</code>&lt;br&gt;<code>ora_dict_obj_type</code></td>
</tr>
<tr>
<td>AFTER RENAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE REVOKE</td>
<td>When a revoke statement is issued</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_dict_obj_name</code>&lt;br&gt;<code>ora_dict_obj_type</code>&lt;br&gt;<code>ora_dict_obj_owner</code>&lt;br&gt;<code>ora_revokee</code>&lt;br&gt;<code>ora_privileges</code></td>
</tr>
<tr>
<td>AFTER REVOKE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER SUSPEND</td>
<td>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.</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_server_error</code>&lt;br&gt;<code>ora_is_servererror</code>&lt;br&gt;<code>space_error_info</code></td>
</tr>
<tr>
<td>BEFORE TRUNCATE</td>
<td>When an object is truncated</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_dict_obj_name</code>&lt;br&gt;<code>ora_dict_obj_type</code>&lt;br&gt;<code>ora_dict_obj_owner</code></td>
</tr>
<tr>
<td>AFTER TRUNCATE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This chapter explains how to bundle related PL/SQL code and data into a package. A package is compiled and stored in the database, where many applications can share its contents.

Topics:
- What is a 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 a package spec, use the CREATE PACKAGE Statement on page 14-39. To create a package body, use the CREATE PACKAGE BODY Statement on page 14-42.

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
What Goes in a PL/SQL Package?

A PL/SQL package contains the following:

- 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 must change a query that is used in many places.

- Declarations for exceptions. Typically, you must be able to reference these from different subprograms, so that you can handle exceptions within invoked subprograms.

- Declarations for subprograms that invoke each other. You need not worry about compilation order for packaged subprograms, making them more convenient than standalone stored subprograms when they invoke back and forth to each other.

- Declarations for overloaded subprograms. You can create multiple variations of a subprogram, using the same names but different sets of parameters.

- Variables 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 more information, see CREATE PACKAGE Statement on page 14-39. For an examples of a PL/SQL packages, see Example 1–19 on page 1-21 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, the database does not recompile other subprograms that invoke the function; these subprograms only depend on the parameters and return value that are declared in the 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.
Referencing PL/SQL Package Contents

Figure 10–1  Package Scope

The spec lists the package resources available to applications. All the information your application must 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`:

```oracle
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

That is all the information needed to invoke the function. You need not consider its underlying implementation (whether it is iterative or recursive for example).

If a 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

```oracle
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 1–20 on page 1-22 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.

```sql
EXEC SQL EXECUTE
BEGIN
  emp_actions.hire_employee(:emp_id,:emp_lname,:emp_fname, ...);
END;
```

**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 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

```sql
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 the database for use by any application that has execute privilege on the package.

Example 10–3 Creating the emp_admin Package

```sql
-- 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,
    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 data types. 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 need not qualify references to its contents by prefixing the package name. For example, you might invoke ABS from a database trigger, stored subprogram, Oracle tool, or 3GL application, as follows:

abs_diff := ABS(x - y);

If you declare your own version of ABS, your local declaration overrides the global declaration. You can still invoke the built-in function by specifying its full name:

abs_diff := STANDARD.ABS(x - y);

Most built-in functions are overloaded. For example, package STANDARD contains 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;
Overview of Product-Specific PL/SQL Packages

Various Oracle tools are supplied with product-specific packages that define application programming interfaces (APIs) that you can invoke from PL/SQL, SQL, Java, and other programming environments. This section briefly describes 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 about these and other product-specific packages, see Oracle Database PL/SQL Packages and Types Reference.

DBMS_ALERT Package

DBMS_ALERT package lets you use database triggers to alert an application when specific database values change. The alerts are transaction based and asynchronous (that is, they operate independently of any timing mechanism). For example, a company might use this package to update the value of its investment portfolio as 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 the database.

DBMS_CONNECTION_POOL Package

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

See Also:
- Oracle Database PL/SQL Packages and Types Reference for a detailed description of the DBMS_CONNECTION_POOL package
- Oracle Database Administrator's Guide for information about managing the Database Resident Connection Pool

HTF and HTP Packages

HTF and HTP packages enable your PL/SQL programs to generate HTML tags.

UTL_FILE Package

UTL_FILE package lets PL/SQL programs read and write operating system text files. It provides a restricted version of standard operating system stream file I/O, including open, put, get, and close operations.

When you want to read or write a text file, you invoke the function FOPEN, which returns a file handle for use in subsequent subprogram calls. For example, the procedure PUT_LINE writes a text string and line terminator to an open file, and the procedure GET_LINE reads a line of text from an open file into an output buffer.

UTL_HTTP Package

UTL_HTTP package enables your PL/SQL programs to make hypertext transfer protocol (HTTP) callouts. It can retrieve data from the Internet or invoke Oracle Web Server cartridges. The package has multiple entry points, each of which accepts a URL (uniform resource locator) string, contacts the specified site, and returns the requested data, which is usually in hypertext markup language (HTML) format.

UTL_SMTP Package

UTL_SMTP package enables your PL/SQL programs to send electronic mails (e-mails) over Simple Mail Transfer Protocol (SMTP). The package provides interfaces to the SMTP commands for an e-mail client to dispatch e-mails to a SMTP server.
Guidelines for Writing PL/SQL Packages

When writing packages, keep them general so they can be reused in future applications. Become familiar with the packages that Oracle supplies, 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 the database 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") 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 about the cursor syntax, see Explicit Cursor on page 13-53.

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

```sql
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 data type 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 data types of items in the SELECT list and the RETURN clause must match.

Packaged cursors increase flexibility. For example, you can change the cursor body in the last example, without having to change the cursor spec.

From a PL/SQL block or subprogram, you use dot notation to reference a packaged cursor, as the following example shows:

```sql
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

PL/SQL run-time errors can arise from design faults, coding mistakes, hardware failures, and many other sources. You cannot anticipate all possible errors, but you can code exception handlers that allow your program to continue to operate 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
- 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. An exception can be either internally defined (by the run-time system) or user-defined. Examples of internally defined exceptions are ORA-22056 (value string is divided by zero) and ORA-27102 (out of memory). Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

You can define your own exceptions in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named insufficient_funds to flag overdrawn bank accounts. User-defined exceptions must be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements or invocations of the procedure DBMS_STANDARD.Raise_Application_Error.
To handle raised exceptions, you write separate 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 about managing errors when using BULK COLLECT, see Handling FORALL Exceptions (%BULK_EXCEPTIONS Attribute) on page 12-16.

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**

```plsql
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
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Company must have had zero earnings.');
    BEGIN
        pe_ratio := NULL;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Some other kind of error occurred.');
    BEGIN
        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.

```plsql
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
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, as in Example 11–2.

Example 11–2  Managing Multiple Errors with a Single Exception Handler

```sql
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';
```

hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still must take corrective action.

- Add error-checking code whenever bad input data can cause an error.
  Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.
  Test your code with different combinations of bad data to see what potential errors arise.

- Make your programs robust enough to work even if the database is not in the state you expect.
  For example, perhaps a table you query will have columns added or deleted, or their types changed. You can avoid such problems by declaring individual variables with %TYPE qualifiers, and declaring records to hold query results with %ROWTYPE qualifiers.

- Handle named exceptions whenever possible, instead of using WHEN OTHERS in exception handlers.
  Learn the names and causes of the predefined exceptions. If your database operations might cause particular ORA-n errors, associate names with these errors so you can write handlers for them. (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.
Instead of checking for an error at every point where it might occur, add an exception handler to your PL/SQL block. If the exception is ever raised in that block (including inside a sub-block), it 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 must check for errors at a specific spot, you can enclose a single statement or a group of statements inside its own `BEGIN-END` block with its own exception handler.

Isolating error-handling routines makes the rest of the program easier to read and understand.

### Predefined PL/SQL Exceptions

An internal exception is raised automatically if your PL/SQL program violates a database rule or exceeds a system-dependent limit. PL/SQL predefines some common ORA-\*n errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

You can use the pragma `EXCEPTION_INIT` to associate exception names with other Oracle Database error codes that you can anticipate. To handle unexpected Oracle Database errors, you can use the `OTHERS` handler. Within this handler, you can invoke the functions `SQLCODE` and `SQLERRM` to return the Oracle Database error code and message text. Once you know the error code, you can use it with pragma `EXCEPTION_INIT` and write a handler specifically for that error.

PL/SQL declares predefined exceptions globally in package `STANDARD`. You need not declare them yourself. You can write handlers for predefined exceptions using the names in Table 11–1.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>ORA Error</th>
<th>SQLCODE</th>
<th>Raised When ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS INTO NULL</td>
<td>06530</td>
<td>-6530</td>
<td>A program attempts to assign values to the attributes of an uninitialized object</td>
</tr>
<tr>
<td>CASE NOT FOUND</td>
<td>06592</td>
<td>-6592</td>
<td>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.</td>
</tr>
<tr>
<td>COLLECTION IS NULL</td>
<td>06531</td>
<td>-6531</td>
<td>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.</td>
</tr>
<tr>
<td>CURSOR ALREADY OPEN</td>
<td>06511</td>
<td>-6511</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 11–1 (Cont.) Predefined PL/SQL Exceptions

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>ORA Error</th>
<th>SQLCODE</th>
<th>Raised When ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP_VAL_ON_INDEX</td>
<td>00001</td>
<td>-1</td>
<td>A program attempts to store duplicate values in a column that is constrained by a unique index.</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>01001</td>
<td>-1001</td>
<td>A program attempts a cursor operation that is not allowed, such as closing an unopened cursor.</td>
</tr>
<tr>
<td>INVALID_NUMBER</td>
<td>01722</td>
<td>-1722</td>
<td>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, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-clause expression in a bulk FETCH statement does not evaluate to a positive number.</td>
</tr>
<tr>
<td>LOGIN_DENIED</td>
<td>01017</td>
<td>-1017</td>
<td>A program attempts to log on to the database with an invalid username or password.</td>
</tr>
<tr>
<td>NO_DATA_FOUND</td>
<td>01403</td>
<td>+100</td>
<td>A SELECT INTO 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.</td>
</tr>
<tr>
<td>NOT_LOGGED_ON</td>
<td>01012</td>
<td>-1012</td>
<td>A program issues a database call without being connected to the database.</td>
</tr>
<tr>
<td>PROGRAM_ERROR</td>
<td>06501</td>
<td>-6501</td>
<td>PL/SQL has an internal problem.</td>
</tr>
<tr>
<td>ROWTYPE_MISMATCH</td>
<td>06504</td>
<td>-6504</td>
<td>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.</td>
</tr>
<tr>
<td>SELF_IS_NULL</td>
<td>30625</td>
<td>-30625</td>
<td>A program attempts to invoke a MEMBER method, but the instance of the object type was not initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method.</td>
</tr>
<tr>
<td>STORAGE_ERROR</td>
<td>06500</td>
<td>-6500</td>
<td>PL/SQL ran out of memory or memory was corrupted.</td>
</tr>
<tr>
<td>SUBSCRIPT_BEYOND_COUNT</td>
<td>06533</td>
<td>-6533</td>
<td>A program references a nested table or varray element using an index number larger than the number of elements in the collection.</td>
</tr>
<tr>
<td>SUBSCRIPT_OUTSIDE_LIMIT</td>
<td>06532</td>
<td>-6532</td>
<td>A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.</td>
</tr>
<tr>
<td>SYS_INVALID_ROWID</td>
<td>01410</td>
<td>-1410</td>
<td>The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.</td>
</tr>
<tr>
<td>TIMEOUT_ON_RESOURCE</td>
<td>00051</td>
<td>-51</td>
<td>A time out occurs while the database is waiting for a resource.</td>
</tr>
</tbody>
</table>
Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike a predefined exception, a user-defined exception must be declared and then raised explicitly, using either a `RAISE` statement or the procedure `DBMS_STANDARD.RAISE_APPLICATION_ERROR`. The latter lets you associate an error message with the user-defined exception.

Topics:
- Declaring PL/SQL Exceptions
- Scope Rules for PL/SQL Exceptions
- Associating a PL/SQL Exception with a Number (EXCEPTION_INITPragma)
- 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`:

```plsql
DECLARE
    past_due EXCEPTION;
END;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

### 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`.

Example 11–3 illustrates the scope rules.

**Example 11–3 Scope of PL/SQL Exceptions**

```sql
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.');</n
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Could not recognize PAST_DUE_EXCEPTION in this scope.');</n
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 (EXCEPTION_INIT Pragma)**

To handle error conditions (typically ORA-`n` 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 Database error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma `EXCEPTION_INIT` in the declarative part of a PL/SQL block, subprogram, or package using the following syntax:

```sql
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

where `exception_name` is the name of a previously declared exception and the number is a negative value corresponding to an ORA-`n` error. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in Example 11–4.
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-n error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To invoke RAISE_APPLICATION_ERROR, use 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 need not 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 Database 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 need not qualify RAISE_APPLICATION_ERROR with DBMS_STANDARD */
        RAISE_APPLICATION_ERROR
            (-20101, 'Expecting at least 1000 tables');
    ELSE
        -- Do rest of processing (for nonerror case)
        NULL;
    END IF;
END;
/
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 Database error number using EXCEPTION_INIT. Other user-defined exceptions must be raised explicitly, with either RAISE statements or invocations of the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

Raise an exception in a PL/SQL block or subprogram only when an error makes it undesirable or impossible to finish processing. You can explicitly raise a given exception anywhere 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 Raise a User-Defined Exception

DECLARE
  out_of_stock   EXCEPTION;
  number_on_hand NUMBER := 0;
```

Handling PL/SQL Errors 11-9
BEGIN
  IF number_on_hand < 1 THEN
    RAISE out_of_stock; -- raise an exception that you 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 Raise a Predefined Exception

DECLARE
  acct_type INTEGER := 7;
BEGIN
  IF acct_type NOT IN (1, 2, 3) THEN
    RAISE INVALID_NUMBER;  -- raise predefined exception
  END IF;
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('HANDLING INVALID INPUT BY ROLLING BACK.');
    ROLLBACK;
END;
/

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

BEGIN
BEGIN
   IF X = 1 THEN
      RAISE A;
   ELSIF X = 2 THEN
      RAISE B;
   ELSE
      RAISE C;
   END IF;
   ...
END;

EXCEPTION
   WHEN A THEN
      ...
END;

BEGIN
EXCEPTION
   WHEN B THEN
      ...
END;

Exception A is handled locally, then execution resumes in the enclosing block

Figure 11–2 Propagation Rules: Example 2

BEGIN
BEGIN
   IF X = 1 THEN
      RAISE A;
   ELSIF X = 2 THEN
      RAISE B;
   ELSE
      RAISE C;
   END IF;
   ...
END;

EXCEPTION
   WHEN A THEN
      ...
END;

BEGIN
EXCEPTION
   WHEN B THEN
      ...
END;

Exception B propagates to the first enclosing block with an appropriate handler

Exception B is handled, then control passes to the host environment
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

```sql
BEGIN
  DECLARE  ---------- sub-block begins
    past_due EXCEPTION;
    due_date DATE := trunc(SYSDATE) - 1;
    todays_date DATE := trunc(SYSDATE);
  BEGIN
    IF due_date < todays_date THEN
      RAISE past_due;
    END IF;
  END;
  ------------- sub-block ends
  EXCEPTION
    WHEN OTHERS THEN
      ROLLBACK;
  END;
/
```

Because the block that declares the exception `past_due` has no handler for it, the exception propagates to the enclosing block. But the enclosing block cannot reference the name `PAST_DUE`, because the scope where it was declared no longer exists. Once the exception name is lost, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the invoking application gets ORA-06510.

## Reraising 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, as in Example 11–9.
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
- 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 declaration in Example 11–10 raises an exception because the constant `credit_limit` cannot store numbers larger than 999.

**Example 11–10  Raising an Exception in a Declaration**

```plsql
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:

```plsql
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**

In an exception handler, you can retrieve the error code with the built-in function `SQLCODE`. To retrieve the associated error message, you can use either the packaged function `DBMS_UTIL.TYPER_FORMAT_ERROR_STACK` or the built-in function `SQLERRM`.

`SQLERRM` returns a maximum of 512 bytes, which is the maximum length of an Oracle Database error message (including the error code, nested messages, and message inserts, such as table and column names). `DBMS_UTIL.TYPER_FORMAT_ERROR_STACK` returns the full error stack, up to 2000 bytes. Therefore, `DBMS_UTIL.TYPER_FORMAT_ERROR_STACK` is recommended over `SQLERRM`, except when using the `FORALL` statement with its `SAVE EXCEPTIONS` clause. With `SAVE EXCEPTIONS`, use `SQLERRM`, as in Example 12–9 on page 12-16.

**See Also:**
- `SQLCODE Function` on page 13-125 for syntax and semantics of this function
- `SQLERRM Function` on page 13-126 for syntax and semantics of this function
- Handling `FORALL` Exceptions (%BULK_EXCEPTIONS Attribute) on page 12-16 for information about using the `FORALL` statement with its `SAVE EXCEPTIONS` clause
- Oracle Database PL/SQL Packages and Types Reference for information about `DBMS_UTIL.TYPER_FORMAT_ERROR_STACK`

A SQL statement cannot invoke `SQLCODE` or `SQLERRM`. To use their values in a SQL statement, assign them to local variables first, as in Example 11–11.

**Example 11–11  Displaying SQLCODE and SQLERRM**

```sql
SQL> CREATE TABLE errors (
    2    code NUMBER,
    3    message VARCHAR2(64),
    4    happened TIMESTAMP);
Table created.

SQL> DECLARE
    2    name EMPLOYEES.LAST_NAME%TYPE;
    3    v_code NUMBER;
```
Handling Raised PL/SQL Exceptions

4    v_errm VARCHAR2(64);
5  BEGIN
6    SELECT last_name INTO name
7      FROM EMPLOYEES
8        WHERE EMPLOYEE_ID = -1;
9  EXCEPTION
10    WHEN OTHERS THEN
11        v_code := SQLCODE;
12
13        v_errm := SUBSTR(SQLERRM, 1, 64);
14        DBMS_OUTPUT.PUT_LINE
15          ('Error code ' || v_code || ': ' || v_errm);
16
17        /* Invoke another procedure,
18           declared with PRAGMA AUTONOMOUS_TRANSACTION,
19           to insert information about errors. */
20
21        INSERT INTO errors
22          VALUES (v_code, v_errm, SYSTIMESTAMP);
23  END;
24  /

Error code 100: ORA-01403: no data found

PL/SQL procedure successfully completed.

SQL>

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 irrecoverable 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
**Handling Raised PL/SQL Exceptions**

**Handling PL/SQL Errors**

```
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 Example 11–12, 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 5-29

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 12-16.

**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, change the value that must be unique and continue with the next loop iteration. If the INSERT succeeds, exit from the loop immediately. With this technique, use a FOR or WHILE loop to limit the number of attempts.

**Example 11–13  Retrying a Transaction After an Exception**

```sql
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 must 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**

```sql
CREATE OR REPLACE PROCEDURE loc_var AS
 stmt_no NUMBER;
 name VARCHAR2(100);
BEGIN
 stmt_no := 1;  -- designates 1st SELECT statement
```
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 compilation 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 11–2. You can suppress or display groups of similar warnings during compilation. To refer to all warning messages, use the keyword All.

Table 11–2 PL/SQL Warning Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEVERE</td>
<td>Condition might cause unexpected action or wrong results.</td>
<td>Aliasing problems with parameters</td>
</tr>
<tr>
<td>PERFORMANCE</td>
<td>Condition might cause performance problems.</td>
<td>Passing a VARCHAR2 value to a NUMBER column in an INSERT statement</td>
</tr>
<tr>
<td>INFORMATIONAL</td>
<td>Condition does not affect performance or correctness, but you might want to change it to make the code more maintainable.</td>
<td>Code that can never be executed</td>
</tr>
</tbody>
</table>

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 compilation 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. For more information about PL/SQL compilation parameters, see PL/SQL Units and Compilation Parameters on page 1-25.

**Example 11–15  Controlling the Display of PL/SQL Warnings**

```sql
-- 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 compilation parameters, see PL/SQL Units and Compilation Parameters on page 1-25.

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**

```sql
-- 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;

For more information, see DBMS_WARNING package in Oracle Database PL/SQL Packages
and Types Reference and PLW- messages in Oracle Database Error Messages
This chapter explains how to write efficient new PL/SQL code and speed up existing PL/SQL code.

Topics:

- 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 Units for Native Execution
- Performing Multiple Transformations with Pipelined Table Functions

How PL/SQL Optimizes Your Programs

Prior to Oracle Database Release 10g, the PL/SQL compiler translated your source code to system 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 compilation parameter `PLSQL_OPTIMIZE_LEVEL=1` instead of its default value 2. In even rarer cases, you might see a change in exception action, either an exception that is not raised at all, or one that is raised earlier than expected. Setting `PLSQL_OPTIMIZE_LEVEL=1` prevents the code from being rearranged.

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 \texttt{PLSQL\_OPTIMIZE\_LEVEL} compilation parameter (which is 2) or set it to 3. With \texttt{PLSQL\_OPTIMIZE\_LEVEL=2}, you must specify each subprogram to be inlined. With \texttt{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 \texttt{INLINE} pragma, described in \texttt{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 \texttt{PLSQL\_OPTIMIZE\_LEVEL} compilation parameter
- Oracle Database Reference for information about the static dictionary view \texttt{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 data types \texttt{PLS\_INTEGER}, \texttt{BINARY\_FLOAT}, and \texttt{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 \texttt{INSERT}, \texttt{UPDATE}, or \texttt{DELETE} statements, or looping through query results. You will want to investigate the \texttt{FORALL} statement for issuing DML, and the \texttt{BULK COLLECT INTO} and \texttt{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 \texttt{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 Data Type Conversions
- Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic
- Use BINARY_FLOAT, BINARY_DOUBLE, SIMPLE_FLOAT, and SIMPLE_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**

```plsql
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*. 

---

12-4 Oracle Database PL/SQL Language Reference
Make Loops as Efficient as Possible
Because PL/SQL applications are often built around loops, it is important to optimize both the loop itself and the code inside the loop:

- To issue a series of DML statements, replace loop constructs with FORALL statements.
- To loop through a result set and store the values, use the BULK COLLECT clause on the query to bring the query results into memory in one operation.
- If you must loop through a result set more than once, or issue other queries as you loop through a result set, you can probably enhance the original query to give you exactly the results you want. Some query operators to explore include UNION, INTERSECT, MINUS, and CONNECT BY.
- You can also nest one query inside another (known as a 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.

Regular expression features use characters like '.', '*', '^', and '$' that you might be familiar with from Linux, UNIX, or PERL programming. For multilanguage programming, there are also extensions such as '['lower:']' to match a lowercase letter, instead of '[a-z]' which does not match lowercase accented letters.

Put Least Expensive Conditional Tests First
PL/SQL stops evaluating a logical expression as soon as the result can be determined. This functionality is known as short-circuit evaluation. See Short-Circuit Evaluation on page 2-34.

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 Data Type Conversions
At run time, PL/SQL converts between different data types automatically. For example, assigning a PLS_INTEGER variable to a NUMBER variable results in a conversion because their internal representations are different.

Whenever possible, choose data types carefully to minimize implicit conversions. Use literals of the appropriate types, such as character literals in character expressions and decimal numbers in number expressions.
Minimizing conversions might mean changing the types of your variables, or even working backward and designing your tables with different data types. Or, you might convert data once, such as from an INTEGER column to a PL_S_INTEGER variable, and use the PL/SQL type consistently after that. The conversion from INTEGER to PL_S_INTEGER data type might improve performance, because of the use of more efficient hardware arithmetic. See Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic on page 12-6.

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 data type PLS_INTEGER.
- If the value of the variable will never be NULL, and the variable does not need overflow checking, use the data type SIMPLE_INTEGER.

PLS_INTEGER values use less storage space than INTEGER or NUMBER values, and PLS_INTEGER operations use hardware arithmetic. For more information, see PLS_INTEGER and BINARY_INTEGER Data Types on page 3-2.

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 data type 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, BINARY_DOUBLE, SIMPLE_FLOAT, and SIMPLE_DOUBLE for Floating-Point Arithmetic
The data type 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.

If the value of the variable will never be NULL, use the subtype SIMPLE_FLOAT or BINARY_FLOAT instead of the base type SIMPLE_DOUBLE or BINARY_DOUBLE. Each subtype has the same range as its base type and has a NOT NULL constraint. Without the overhead of checking for nullness, SIMPLE_FLOAT and SIMPLE_DOUBLE provide significantly better performance than BINARY_FLOAT and BINARY_DOUBLE when PLSQL_CODE_TYPE=’NATIVE’, because arithmetic operations on SIMPLE_FLOAT and SIMPLE_DOUBLE values are done directly in the hardware. When PLSQL_CODE_TYPE=’INTERPRETED’, the performance improvement is smaller.

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 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 ages out of memory, and you reference it again, it is reloaded.

You can improve performance by sizing the shared memory pool correctly. Make it large enough to hold all frequently used packages, but not so large that memory is wasted.

Pin Packages in the Shared Memory Pool
You can pin frequently accessed packages in the shared memory pool, using the supplied package DBMS_SHARED_POOL. When a package is pinned, it does not age out; it remains in memory no matter how full the pool gets or how frequently you access the package.

For more information about the DBMS_SHARED_POOL package, see Oracle Database PL/SQL Packages and Types Reference.

Apply Advice of Compiler Warnings
The PL/SQL compiler issues warnings about things that do not make a program incorrect, but might lead to poor performance. If you receive such a warning, and the performance of this code is important, follow the suggestions in the warning and make the code more efficient.

Collecting Data About User-Defined Identifiers
PL/Scope extracts, organizes, and stores data about user-defined identifiers from PL/SQL source code. You can retrieve source code identifier data with the static data dictionary views *_IDENTIFIERS. For more information, see Oracle Database Advanced Application Developer’s Guide.
Profiling and Tracing PL/SQL Programs

To help you isolate performance problems in large PL/SQL programs, PL/SQL provides the following tools, implemented as PL/SQL packages:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiler API</td>
<td>DBMS_PROFILER</td>
<td>Computes the time that your PL/SQL program spends at each line and in each subprogram. You must have CREATE privileges on the units to be profiled. Saves run-time statistics in database tables, which you can query.</td>
</tr>
<tr>
<td>Trace API</td>
<td>DBMS_TRACE</td>
<td>Traces the order in which subprograms execute. You can specify the subprograms to trace and the tracing level. Saves run-time statistics in database tables, which you can query.</td>
</tr>
<tr>
<td>PL/SQL hierarchical profiler</td>
<td>DBMS_HPROF</td>
<td>Reports the dynamic execution program profile of your PL/SQL program, organized by subprogram calls. Accounts for SQL and PL/SQL execution times separately. Requires no special source or compile-time preparation. Generates reports in HTML. Provides the option of storing results in relational format in database tables for custom report generation (such as third-party tools offer).</td>
</tr>
</tbody>
</table>

**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 12-8).

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 results to PL/SQL. You can minimize the performance overhead of this communication between PL/SQL and SQL by using the PL/SQL features that are known collectively as bulk SQL.

The FORALL statement sends INSERT, UPDATE, or DELETE statements in batches, rather than one at a time. The BULK COLLECT clause brings back batches of results from SQL. If the DML statement affects four or more database rows, bulk SQL can improve performance considerably.

Assigning values to PL/SQL variables in SQL statements is called binding. PL/SQL binding operations fall into these categories:
Reducing Loop Overhead for DML Statements and Queries with Bulk SQL

Bulk SQL uses PL/SQL collections to pass large amounts of data back and forth in single operations. This process is called bulk binding. If the collection has $n$ elements, bulk binding uses a single operation to perform the equivalent of $n$ SELECT INTO, INSERT, UPDATE, or DELETE statements. A query that uses bulk binding can return any number of rows, without requiring a FETCH statement for each one.

**Note:** Parallel DML is disabled with bulk binds.

To speed up INSERT, UPDATE, and DELETE statements, enclose the SQL statement within a PL/SQL FORALL statement instead of a LOOP statement.

To speed up SELECT INTO statements, include the BULK COLLECT clause.

Topics:
- Running One DML Statement Multiple Times (FORALL Statement)
- Retrieving Query Results into Collections (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 13-72.

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**

```sql
CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS VARRAY(20) OF NUMBER;
    depts NumList := NumList(10, 30, 70);  -- department numbers
```

---

<table>
<thead>
<tr>
<th>Binding Category</th>
<th>When This Binding Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-bind</td>
<td>When an INSERT or UPDATE statement stores a PL/SQL variable or host variable in the database</td>
</tr>
<tr>
<td>Out-bind</td>
<td>When the RETURNING clause of an INSERT, UPDATE, or DELETE statement assigns a database value to a PL/SQL variable or host variable</td>
</tr>
<tr>
<td>Define</td>
<td>When a SELECT or FETCH statement assigns a database value to a PL/SQL variable or host variable</td>
</tr>
</tbody>
</table>
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**

```sql
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**

```sql
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;
/
```

Reducing Loop Overhead for DML Statements and Queries with Bulk SQL
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**

```sql
-- 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
      -- Delete order i
      DELETE FROM valid_orders
        WHERE cust_name = cust_tab(i); -- order i
      -- Delete order i from both collections
      big_order_tab(i) := NULL;
      rejected_order_tab(i) := NULL;
    END IF;
  END LOOP;
END;
```

/
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 you handle the exception, the first UPDATE is not rolled back and you 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 SQL Cursors (Implicit) 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 ith 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 -- Run 3 DELETE statements.
    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**

```sql
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
        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 deduce 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 predefined error ORA-24381. ORA-24381 is raised if any exceptions are caught and saved after a bulk operation.

All exceptions raised during the execution are saved in the cursor attribute %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 Database 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 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

```
-- 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;

    -- Exception handler for ORA-24381:
    errors    NUMBER;
    dml_errors EXCEPTION;
    PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
    SELECT employee_id
    BULK COLLECT INTO emp_sr FROM emp_temp
```

WHERE hire_date < '30-DEC-94';

-- Add '_SR' to job_id of most senior employees:
FORALL i IN emp_sr.FIRST..emp_sr.LAST SAVE EXCEPTIONS
  UPDATE emp_temp SET job_id = job_id || '_SR'
  WHERE emp_sr(i) = emp_temp.employee_id;
-- If errors occurred during FORALL SAVE EXCEPTIONS,
-- a single exception is raised when the statement completes.

EXCEPTION
-- Figure out what failed and why
WHEN dml_errors THEN
  errors := SQL%BULK_EXCEPTIONS.COUNT;
  DBMS_OUTPUT.PUT_LINE
    ('Number of statements that failed: ' || errors);
  FOR i IN 1..errors LOOP
    DBMS_OUTPUT.PUT_LINE('Error #' || i || ' occurred during '||
      'iteration #' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
    DBMS_OUTPUT.PUT_LINE('Error message is ' || SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
  END LOOP;
END;
/
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 job_id
Error #2 occurred during iteration #13
Error message is ORA-12899: value too large for column job_id

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 Database error message (which includes the code), the value of SQL%BULK_EXCEPTIONS(i).ERROR_CODE was negated and then passed to the error-reporting function SQLERRM, which expects a negative number.

Retrieving Query Results into Collections (BULK COLLECT Clause)

Using the BULK COLLECT clause 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 the BULK COLLECT clause in the SELECT INTO and FETCH INTO statements, and in 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;
  nums 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.
Reducing Loop Overhead for DML Statements and Queries with Bulk SQL

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
- SELECT BULK COLLECT INTO Statements and Aliasing

Examples of Bulk Fetching from a Cursor

You can fetch from a cursor into one or more collections as shown in Example 12–12.

Example 12–12  Bulk-Fetching from a Cursor Into One or More Collections

```sql
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 ---');
  END;
END;
```

Tuning PL/SQL Applications for Performance  12-19
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;
depth_recs DeptRecTab;
CURSOR c1 IS
    SELECT department_id, department_name, manager_id, location_id
      FROM departments
      WHERE department_id > 70;
BEGIN
    OPEN c1;
    FETCH c1 BULK COLLECT INTO dept_recs;
END;
/

Limiting Rows for a Bulk FETCH Operation (LIMIT Clause)

The optional LIMIT clause, allowed only in bulk FETCH statements, limits the number
of rows fetched from the database. In Example 12–14, with each iteration of the loop,
the FETCH statement fetches ten rows (or fewer) 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**

```sql
CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS TABLE OF employees.employee_id%TYPE;
    enums NumList;
    TYPE NameList IS TABLE OF employees.last_name%TYPE;
    names NameList;
BEGIN
    DELETE FROM emp_temp WHERE department_id = 30
    RETURNING employee_id, last_name
    BULK COLLECT INTO enums, names;
    DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
    FOR i IN enums.FIRST .. enums.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE ('Employee #' || enums(i) || ': ' || names(i));
    END LOOP;
END;
/
```

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**

```sql
CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10,20,30);
    TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
    e_ids enum_t;
    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) || ': ' || 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.

```sql
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;
```

### SELECT BULK COLLECT INTO Statements and Aliasing

In a statement of the form

```
SELECT column BULK COLLECT INTO collection FROM table ...
```

*column* and *collection* are analogous to `IN` and `OUT NOCOPY` subprogram parameters, respectively, and PL/SQL passes them by reference. As with subprogram parameters that are passed by reference, aliasing can cause unexpected results.

**See Also:** Understanding PL/SQL Subprogram Parameter Aliasing on page 8-25

In Example 12–17, the intention is to select specific values from a collection, `numbers1`, and then store them in the same collection. The unexpected result is that all elements of `numbers1` are deleted. For workarounds, see Example 12–18 and Example 12–19.

**Example 12–17** SELECT BULK COLLECT INTO Statement with Unexpected Results

```sql
SQL> CREATE OR REPLACE TYPE numbers_type IS
 2    TABLE OF INTEGER
3  /
Type created.

SQL> CREATE OR REPLACE PROCEDURE p (i IN INTEGER) IS
 2    numbers1 numbers_type := numbers_type(1,2,3,4,5);
```
Example 12–18 uses a cursor to achieve the result intended by Example 12–17.

**Example 12–18 Workaround for Example 12–17 Using a Cursor**

```sql
SQL> CREATE OR REPLACE TYPE numbers_type IS
```
TABLE OF INTEGER
/

Type created.

CREATE OR REPLACE PROCEDURE p (i IN INTEGER) IS
  numbers1 numbers_type := numbers_type(1,2,3,4,5);
  CURSOR c IS
    SELECT a.COLUMN_VALUE
    FROM TABLE(numbers1) a
    WHERE a.COLUMN_VALUE > p.i
    ORDER BY a.COLUMN_VALUE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;
  OPEN c;
  FETCH c BULK COLLECT INTO numbers1;
  CLOSE c;
  DBMS_OUTPUT.PUT_LINE('After FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;
END p;
/

Procedure created.

BEGIN
  p(2);
  END;
  /

Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5

After FETCH statement
numbers1.COUNT() = 3
numbers1(1) = 3
numbers1(2) = 4
numbers1(3) = 5

PL/SQL procedure successfully completed.

BEGIN
  p(10);
  END;
Example 12–19 selects specific values from a collection, `numbers1`, and then stores them in a different collection, `numbers2`. Example 12–19 performs faster than Example 12–18.

Example 12–19 Workaround for Example 12–17 Using a Second Collection

```sql
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER;
/
Type created.

CREATE OR REPLACE PROCEDURE p (i IN INTEGER) IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
  numbers2  numbers_type := numbers_type(0,0,0,0,0);
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());
  FOR j IN 1..numbers2.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
  END LOOP;
  SELECT a.COLUMN_VALUE
    BULK COLLECT INTO numbers2      -- numbers2 appears here
    FROM TABLE(numbers1) a        -- numbers1 appears here
    WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;
```
DBMS_OUTPUT.PUT_LINE('After SELECT statement');
DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
        DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
END IF;

DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

IF numbers2.COUNT() > 0 THEN
    FOR j IN 1..numbers2.COUNT() LOOP
        DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
    END LOOP;
END IF;

Procedure created.

SQL> BEGIN
   2   p(2);
   3   END;
   4  /
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0

After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 3
numbers2(1) = 3
numbers2(2) = 4
numbers2(3) = 5

PL/SQL procedure successfully completed.

SQL> BEGIN
   2   p(10);
   3   END;
   4  /
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0

After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 0

PL/SQL procedure successfully completed.

SQL>

Writing Computation-Intensive PL/SQL Programs

The BINARY_FLOAT and BINARY_DOUBLE data types make it practical to write PL/SQL programs to do number-crunching, for scientific applications involving floating-point calculations. These data types act much like the native floating-point types on many hardware systems, with semantics derived from the IEEE-754 floating-point standard.

The way these data types represent decimal data make them less suitable for financial applications, where precise representation of fractional amounts is more important than pure performance.

The PLS_INTEGER data type is a PL/SQL-only data type that is more efficient than the SQL data types NUMBER or INTEGER for integer arithmetic. You can use PLS_INTEGER to write pure PL/SQL code for integer arithmetic, or convert NUMBER or INTEGER values to PLS_INTEGER for manipulation by PL/SQL.

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 12–20 Using NOCOPY with Parameters**

```
Example 12–20 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.
```

```
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 8-25.

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 data type conversion.
The subprogram is called through a database link or as an external subprogram.

Compiling PL/SQL Units for Native Execution

You can usually speed up PL/SQL units by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace.

You can natively compile any PL/SQL unit of any type, including those that Oracle supplies.

Natively compiled program units work in all server environments, including shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

On most platforms, PL/SQL native compilation requires no special set-up or maintenance. On some platforms, the DBA might want to do some optional configuration.

See Also:
- Oracle Database Administrator’s Guide for information about configuring a database
- Platform-specific configuration documentation for your platform

You can test to see how much performance gain you can get by enabling PL/SQL native compilation.

If you have determined that PL/SQL native compilation will provide significant performance gains in database operations, Oracle recommends compiling the entire database for native mode, which requires DBA privileges. This 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 Up a New Database for PL/SQL Native Compilation*
- Compiling the Entire Database for PL/SQL Native or Interpreted Compilation*

* Requires DBA privileges.

Determining Whether to Use PL/SQL Native Compilation

Whether to compile a PL/SQL unit for native or interpreted mode depends on where you are in the development cycle and on what the program unit does.

While you are debugging program units and recompiling them frequently, interpreted mode has these advantages:

- You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).
- Compiling for interpreted mode is faster than compiling for native mode.

After the debugging phase of development, consider the following in determining whether to compile a PL/SQL 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 unit are compiled into an intermediate form, system 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 unit are compiled into native code and stored in the SYSTEM tablespace. The native code need not be interpreted at run time, so it runs faster.

Because native compilation applies only to PL/SQL statements, a PL/SQL unit that only calls SQL statements might not run faster when natively compiled, but it 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 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 it. If a program unit is not being used, the shared memory it is using might be freed, to reduce memory load.

Natively compiled subprograms and interpreted subprograms can call each other.

PL/SQL native compilation works transparently in a Oracle Real Application Clusters (Oracle RAC) environment.

The `PLSQL_CODE_TYPE` compilation parameter determines whether PL/SQL code is natively compiled or interpreted. For information about this compilation parameters, see PL/SQL Units and Compilation Parameters on page 1-25.

Dependencies, Invalidation, and Revalidation

Recompilation is automatic with invalidated PL/SQL modules. For example, if an object on which a natively compiled PL/SQL subprogram depends changes, the subprogram is invalidated. The next time the same subprogram is called, the database recompiles the subprogram automatically. Because the `PLSQL_CODE_TYPE` setting is stored inside the library unit for each subprogram, the automatic recompilation uses this stored setting for code type.

Explicit recompilation does not necessarily use the stored `PLSQL_CODE_TYPE` setting. For the conditions under which explicit recompilation uses stored settings, see PL/SQL Units and Compilation Parameters on page 1-25.

Setting Up a New Database for PL/SQL Native Compilation

If you have DBA privileges, you can set up an new database for PL/SQL native compilation by setting the compilation parameter `PLSQL_CODE_TYPE` to `NATIVE`. The performance benefits apply to all the built-in PL/SQL packages, which are used for many database operations.
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 12-30.

**Note:** If you compile the whole database as **NATIVE**, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

During the conversion to native compilation, TYPE specifications are not recompiled by `dbmsupgnv.sql` to **NATIVE** because these specifications do not contain executable code.

Package specifications seldom contain executable code so the run-time benefits of compiling to **NATIVE** are not measurable. You can use the TRUE command-line parameter with the `dbmsupgnv.sql` script to exclude package specs from recompilation to **NATIVE**, saving time in the conversion process.

When converting to interpreted compilation, the `dbmsupgin.sql` script does not accept any parameters and does not exclude any PL/SQL units.

**Note:** The following procedure describes the conversion to native compilation. If you must recompile all PL/SQL modules to interpreted compilation, make these changes in the steps.

- **Note:** If you compile the whole database as **NATIVE**, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

1. Ensure that a test PL/SQL unit can be compiled. For example:
   ```sql
   ALTER PROCEDURE my_proc COMPIL PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS;
   ```

2. Shut down application services, the listener, and the database.
   - Shut down all of the Application services including the Forms Processes, Web Servers, Reports Servers, and Concurrent Manager Servers. After shutting down all of the Application services, ensure that all of the connections to the database were terminated.
   - Shut down the TNS listener of the database to ensure that no new connections are made.
   - Shut down the database in normal or immediate mode as the user `SYS`. See the Oracle Database Administrator's Guide.

3. Set `PLSQL_CODE_TYPE` to **NATIVE** in the compilation parameter file. If the database is using a server parameter file, then set this after the database has started.

- **Note:** If you compile the whole database as **NATIVE**, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.
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 about 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:

```sql
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):

```sql
SELECT TYPE, PLSQL_CODE_TYPE, COUNT(*)
FROM DBA_PLSQL_OBJECT_SETTINGS
WHERE PLSQL_CODE_TYPE IS NOT NULL
GROUP BY TYPE, PLSQL_CODE_TYPE
ORDER BY TYPE, PLSQL_CODE_TYPE;
```

Any objects with a NULL plsql_code_type are special internal objects and can be ignored.

7. Run the $ORACLE_HOME/rdbms/admin/dbmsupgnv.sql script as the user SYS to update the plsql_code_type setting to NATIVE in the dictionary tables for all PL/SQL units. This process also invalidates the units. Use TRUE with the script to exclude package specifications; FALSE to include the package specifications.

This update must be done when the database is in UPGRADE mode. The script is guaranteed to complete successfully or rollback all the changes.

8. Shut down the database and restart in NORMAL mode.

9. Before you run the utlrp.sql script, Oracle recommends that no other sessions are connected to avoid possible problems. You can ensure this with the following statement:

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;
```

10. Run the $ORACLE_HOME/rdbms/admin/utlrp.sql script as the user SYS. This script recompiles all the PL/SQL modules using a default degree of parallelism. See the comments in the script for information about setting the degree explicitly.

If for any reason the script is abnormally terminated, rerun the utlrp.sql script to recompile any remaining invalid PL/SQL modules.

11. After the compilation completes successfully, verify that there are no 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. Reexecute the query in step 6. If recompiling with dbmsupgnv.sql, confirm that all PL/SQL units, except TYPE specifications and package specifications if excluded, are NATIVE. If recompiling with dbmsupgin.sql, confirm that all PL/SQL units are INTERPRETED.

13. Disable the restricted session mode for the database, then start the services that you previously shut down. To disable restricted session mode, use the following statement:

   ALTER SYSTEM DISABLE RESTRICTED SESSION;

Performing Multiple Transformations with Pipelined Table Functions

This section explains 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.

Note: A pipelined table function cannot be run over a database link. The reason is that the return type of a pipelined table function is a SQL user-defined type, which can be used only within a single database (as explained in Oracle Database Object-Relational Developer’s Guide). Although the return type of a pipelined table function might appear to be a PL/SQL type, the database actually converts that PL/SQL type to a corresponding SQL user-defined type.

Topics:

- Overview of Pipelined Table Functions
- Writing a Pipelined Table Function
- 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 let you use PL/SQL to program a row source. You invoke the table function as the operand of the table operator in the FROM list of a SQL SELECT statement. It is also possible to invoke a table function as a SELECT list item; here, you do not use the table operator.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type (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.

**Note:** When rows from a collection returned by a table function are pipelined, the pipelined function always references the current state of the data. After opening the cursor on the collection, if the data in the collection is changed, then the change is reflected in the cursor. PL/SQL variables are private to a session and are not transactional. Therefore, the notion of read-consistency, well known for its applicability to table data, does not apply to PL/SQL collection variables.

Streaming, pipelining, and parallel execution of table functions can improve performance:

- By enabling multithreaded, concurrent execution of table functions
- By eliminating intermediate staging between processes
- By improving query response time: With non-pipelined table functions, the entire collection returned by a table function must be constructed and returned to the server before the query can return a single result row. Pipelining enables rows to be returned iteratively, as they are produced. This also reduces the memory that a table function requires, as the object cache need not materialize the entire collection.
- By iteratively providing result rows from the collection returned by a table function as the rows are produced instead of waiting until the entire collection is staged in tables or memory and then returning the entire collection.

**Writing a Pipelined Table Function**

You declare a pipelined table function by specifying the PIPELINED keyword. Pipelined functions can be defined at the schema level with CREATE FUNCTION or in a package. The PIPELINED keyword indicates that the function returns rows iteratively. The return type of the pipelined table function must be a supported collection type, such as a nested table or a varray. This collection type can be declared at the schema level or inside a package. Inside the function, you return individual elements of the collection type. The elements of the collection type must be supported SQL data types, such as NUMBER and VARCHAR2. PL/SQL data types, such as PLS_INTEGER and BOOLEAN, are not supported as collection elements in a pipelined function.

**Example 12–21** 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 12–21 Assigning the Result of a Table Function**

```sql
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)
```

Tuning PL/SQL Applications for Performance 12-35
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 12–22, the f_trans function converts a row of the employees table into two rows.

Example 12–22 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;
In the preceding query, the pipelined table function \texttt{f\_trans} fetches rows from the \texttt{CURSOR} subquery \texttt{SELECT * FROM employees WHERE department\_id = 60)}, performs the transformation, and pipelines the results back to the user as a table. The function produces two output rows (collection elements) for each input row.

When a \texttt{CURSOR} subquery is passed from SQL to a \texttt{REF CURSOR} function argument as in Example 12–22, the referenced cursor is already open when the function begins executing.

### Returning Results from Pipelined Table Functions

In PL/SQL, the \texttt{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 12–22, the \texttt{PIPE ROW(out\_rec)} statement pipelines data out of the PL/SQL table function. \texttt{out\_rec} is a record, and its type matches the type of an element of the output collection.

The \texttt{PIPE ROW} statement may be used only in the body of pipelined table functions; an exception is raised if it is used anywhere else. The \texttt{PIPE ROW} statement can be omitted for a pipelined table function that returns no rows.

A pipelined table function may have a \texttt{RETURN} statement that does not return a value. The \texttt{RETURN} statement transfers the control back to the consumer and ensures that the next fetch gets a \texttt{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 \texttt{PRAGMA AUTONOMOUS\_TRANSACTION}. If a table function is part of an autonomous transaction, it must \texttt{COMMIT} or \texttt{ROLLBACK} before each \texttt{PIPE ROW} statement, to avoid an error in the calling subprogram.

The database has three special SQL data types that enable you to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these three special types to create unnamed types, including anonymous collection types. The types are \texttt{SYS.ANYTYPE}, \texttt{SYS.ANYDATA}, and \texttt{SYS.ANYDATASET}. The \texttt{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 \texttt{ANYTYPE}, \texttt{ANYDATA}, and \texttt{ANYDATASET} types and about the \texttt{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 \texttt{g} to function \texttt{f}:

\[
\text{SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));}
\]
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:

```sql
SELECT * FROM TABLE(f(...)) t1, TABLE(f(...)) t2
WHERE t1.id = t2.id;
SELECT * FROM TABLE(f());
SELECT * FROM TABLE(f());
```

If the function always produces the same result value for each combination of values passed in, you can declare the function `DETERMINISTIC`, and the database automatically buffers rows for it. If the function is not really deterministic, results are unpredictable.

### Fetching from Results of Pipelined Table Functions

PL/SQL cursors and ref cursors can be defined for queries over table functions. For example:

```sql
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:

```sql
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;
/
```

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`:

```sql
FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;
```

Results of a subquery can be passed to a function directly:

```sql
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 need not first create a REF CURSOR type in a package before you can use it.

To use a strong REF CURSOR type, you still must create a PL/SQL package and declare a strong REF CURSOR type in it. Also, if you are using a strong REF CURSOR type as an argument to a table function, then the actual type of the 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 12–23.

For more information about cursor variables, see Declaring REF CURSOR Types and Cursor Variables on page 6-23.

**Example 12–23 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);
    END LOOP;
  END g_trans;
END refcur_pkg;
/
```
PIPE ROW(out_rec);
END LOOP;
CLOSE p1;
RETURN;
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 12–24 computes a weighted average by iterating over a set of input rows.

**Example 12–24 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 exception:

UPDATE F(CURSOR(SELECT * FROM tab)) SET col = value;
INSERT INTO f(...) VALUES ('any', 'thing');

However, you can create a view over a table function and use INSTEAD OF triggers to update it. For example:

CREATE VIEW BookTable AS SELECT x.Name, x.Author
FROM TABLE(GetBooks('data.txt')) x;

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.
This chapter summarizes the syntax and semantics of PL/SQL language elements and provides links to examples and related topics.

For instructions for reading the syntax diagrams in this chapter, see Oracle Database SQL Language Reference.

Topics:
- Assignment Statement
- AUTONOMOUS_TRANSACTION Pragma
- Block
- CASE Statement
- CLOSE Statement
- Collection
- Collection Method Call
- Comment
- Constant
- CONTINUE Statement
- Cursor Attribute
- Cursor Variable Declaration
- EXCEPTION_INIT Pragma
- Exception Declaration
- Exception Handler
- EXECUTE IMMEDIATE Statement
- EXIT Statement
- Explicit Cursor
- Expression
- FETCH Statement
- FORALL Statement
- Function Declaration and Definition
- GOTO Statement
- IF Statement
- INLINE Pragma
- Literal
- LOOP Statements
- NULL Statement
- OPEN Statement
- OPEN-FOR Statement
- Parameter Declaration
- Procedure Declaration and Definition
- RAISE Statement
- Record Definition
- RESTRICT_REFERENCES Pragma (deprecated)
- RETURN Statement
- RETURNING INTO Clause
- %ROWTYPE Attribute
- SELECT INTO Statement
- SERIALLY_REUSABLE Pragma
- SQL ( Implicit) Cursor Attribute
- SQLCODE Function
- SQLERRM Function
- %TYPE Attribute
- Variable
Assignment Statement

The assignment statement sets the current value of a variable, field, parameter, or element that has been declared in the current scope.

The assignment operator (:=) in the assignment statement can also appear in a constant or variable declaration. In a variable declaration, it assigns a default value to the variable. Without a default value, a variable is initialized to NULL every time a block is entered.

If a variable does not have a default value, always use the assignment statement to assign a value to it before using it in an expression.

Syntax

assignment_statement ::= 

Keyword and Parameter Descriptions

attribute_name
The name of an attribute of object_type. The name must be unique within object_type (but can be used in other object types).

You cannot initialize an attribute in its declaration. You cannot impose the NOT NULL constraint on an attribute.

See Also: CREATE TYPE Statement on page 14-65 for information about attributes of object types

collection_name
The name of a collection.

cursor_variable_name
The name of a PL/SQL cursor variable.
**expression**

The expression whose value is to be assigned to the target (the item to the left of the assignment operator) when the assignment statement executes.

The value of expression must have a data type that is compatible with the data type of the target.

If the target is a variable defined as NOT NULL, the value of expression cannot be NULL. If the target is a Boolean variable, the value of expression must be TRUE, FALSE, or NULL. If the target is a cursor variable, the value of expression must also be a cursor variable.

**field_name**

The name of a field in record_name.

Specify field_name if you want to assign the value of expression to a specific field of a record.

Omit field_name if you want to assign the value of expression to all fields of record_name at once; that is, if you want to assign one record to another. You can assign one record to another only if their declarations refer to the same table or cursor, as in Example 2–17, "Assigning One Record to Another, Correctly and Incorrectly" on page 2-16.

**host_cursor_variable_name**

The name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

**host_variable_name**

The name of a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument.

**index**

A numeric expression whose value has data type PLS_INTEGER or a data type implicitly convertible to PLS_INTEGER (see Table 3–10, "Possible Implicit PL/SQL Data Type Conversions" on page 3-31).

Specify index if you want to assign the value of expression to a specific element of collection_name.

Omit index if you want to assign the value of expression to all elements of collection_name at once; that is, if you want to assign one collection to another. You can assign one collection to another only if the collections have the same data type (not merely the same element type).

**indicator_name**

The name of an indicator variable for host_variable_name.

An indicator variable indicates the value or condition of its host variable. For example, in the Oracle Precompiler environment, indicator variables let you detect NULL or truncated values in output host variables.
**object_name**
The name of an instance of an object type.

**parameter_name**
The name of a formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears.

**record_name**
The name of a user-defined or %ROWTYPE record.

**variable_name**
The name of a PL/SQL variable.

### Examples
- Example 1–3, "Assigning Values to Variables with the Assignment Operator" on page 1-7
- Example 2–17, "Assigning One Record to Another, Correctly and Incorrectly" on page 2-16
- Example 2–30, "Assigning BOOLEAN Values" on page 2-27
- Example 3–4, "Assigning a Literal Value to a TIMESTAMP Variable" on page 3-17
- Example 5–17, "Data Type Compatibility for Collection Assignment" on page 5-14

### Related Topics
- Constant on page 13-32
- Expression on page 13-57
- Variable on page 13-130
- SELECT INTO Statement on page 13-116
- Assigning Values to Variables on page 2-26
- Assigning Values to Collections on page 5-13
- Assigning Values to Records on page 5-34
AUTONOMOUS_TRANSACTIONPragma

**AUTONOMOUS_TRANSACTION Pragma**

The `AUTONOMOUS_TRANSACTION` pragma marks a routine as autonomous; that is, independent of the main transaction.

In this context, a routine is one of the following:

- Top-level (not nested) anonymous PL/SQL block
- Standalone, packaged, or nested subprogram
- Method of a SQL object type
- Database trigger

When an autonomous routine is invoked, the main transaction is suspended. The autonomous transaction is fully independent of the main transaction: they share no locks, resources, or commit dependencies. The autonomous transaction does not affect the main transaction.

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. They become visible to the main transaction when it resumes only if its isolation level is `READ COMMITTED` (the default).

**Syntax**

```plaintext
autonomous_transactionPragma ::= \\
PRAGMA AUTONOMOUS_TRANSACTION ;
```

**Keyword and Parameter Descriptions**

**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.

**AUTONOMOUS_TRANSACTION**

Signifies that the routine is autonomous.

**Usage Notes**

You cannot apply this pragma to an entire package, but you can apply it to each subprogram in a package.

You cannot apply this pragma to an entire an object type, but you can apply it to each method of a SQL object type.

Unlike an ordinary trigger, an autonomous trigger can contain transaction control statements, such as `COMMIT` and `ROLLBACK`, and can issue DDL statements (such as `CREATE` and `DROP`) through the `EXECUTE IMMEDIATE` statement.

In the main transaction, rolling back to a savepoint located before the call to the autonomous subprogram does not roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.
If an autonomous transaction attempts to access a resource held by the main transaction (which cannot resume until the autonomous routine exits), a deadlock can occur. The database raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception goes unhandled, or if the transaction ends because of some other unhandled exception, the transaction is rolled back.

You cannot execute a `PIPE ROW` statement in your autonomous routine while your autonomous transaction is open. You must close the autonomous transaction before executing the `PIPE ROW` statement. This is normally accomplished by committing or rolling back the autonomous transaction before executing the `PIPE ROW` statement.

**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
- Example 6–48, "Invoking an Autonomous Function" on page 6-46

**Related Topics**

- EXCEPTION_INIT Pragma on page 13-43
- INLINE Pragma on page 13-82
- RESTRICT_REFERENCES Pragma on page 13-107
- SERIALLY_REUSABLE Pragma on page 13-120
- Doing Independent Units of Work with Autonomous Transactions on page 6-40
The block, which groups related declarations and statements, is the basic unit of a PL/SQL source program. It has an optional declarative part, a required executable part, and an optional exception-handling part. Declarations are local to the block and cease to exist when the block completes execution.

A block can appear either at schema level (as a top-level block) or inside another block (as a nested block). A block can contain another block wherever it can contain an executable statement.

**Syntax**

\[
\text{plsql\_block ::=}
\]

- \( \text{body} \) (on page 13-11)
- \( \text{declare\_section} \) (on page 13-9)
- \( \text{item\_list\_1} \) (on page 13-10)
- \( \text{item\_list\_2} \) (on page 13-9)

**Example Syntax**

\[
\text{<< label_name >>} \quad \text{DECLARE} \quad \text{declare\_section} \quad \text{body}
\]

\[
\text{(body := on page 13-11)}
\]

**Declare Section**

\[
\text{declare\_section ::=}
\]

- \( \text{item\_list\_1} \)
- \( \text{item\_list\_2} \)

\[
\text{(item\_list\_2 := on page 13-9)}
\]

**Item List 1**

\[
\text{item\_list\_1 ::=}
\]

- \( \text{type\_definition} \)
- \( \text{item\_declaration} \)
- \( \text{function\_declaration} \)
- \( \text{procedure\_declaration} \)
- \( \text{pragma} \)

\[
\text{(type\_definition := on page 13-9, item\_declaration := on page 13-10, function\_declaration := on page 13-75, procedure\_declaration := on page 13-101, pragma := on page 13-10)}
\]
item_list_2 ::= (function_declaration ::= on page 13-75, function_definition ::= on page 13-75, procedure_declaration ::= on page 13-101, procedure_definition ::= on page 13-101, pragma ::= on page 13-10)

type_definition ::= (record_type_definition ::= on page 13-104, ref_cursor_type_definition ::= on page 13-39, collection_type_definition ::= on page 13-22)

subtype_definition ::= (SUBTYPE {constraint} NOT NULL subtype_name IS base_type)

declaration ::= on page 13-44, record_type_declaration ::= on page 13-104, variable_declarion ::= on page 13-130

**pragma ::=**

- autonomous_transaction_pragma
- exception_init_pragma
- inline_pragma
- restrict_references_pragma
- serially_resuable_pragma

 aunonomous_transaction_pragma ::= on page 13-6, exception_init_pragma ::= on page 13-43, inline_pragma ::= on page 13-82, restrict_references_pragma ::= on page 13-107, serially_resuable_pragma ::= on page 13-120

**body ::=**

- BEGIN statement pragma
- EXCEPTION exception_handler END name

exception_handler ::= on page 13-45
statement ::= 

sql_statement ::= 

Keyword and Parameter Descriptions

base_type
Any scalar or user-defined PL/SQL data type 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).

**collection_variable_dec**
Declares a collection (index-by table, nested table, or varray). For the syntax of `collection_declaration`, see Collection on page 13-22.

**constant_declaration**
Declares a constant. For the syntax of `constant_declaration`, see Constant on page 13-32.

**constraint**
Applies only to data types that can be constrained such as `CHAR` and `NUMBER`. For character data types, this specifies a maximum size in bytes. For numeric data types, this specifies a maximum precision and scale.

**cursor_declaration**
Declares an explicit cursor. For the syntax of `cursor_declaration`, see Explicit Cursor on page 13-53.

**cursor_variable_declaration**
Declares a cursor variable. For the syntax of `cursor_variable_declaration`, see Cursor Variable Declaration on page 13-39.

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 2-10.

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 PL/SQL Blocks on page 1-4.

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 PL/SQL Blocks on page 1-4.

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 about exceptions, see Chapter 11, "Handling PL/SQL Errors."

**exception_declaration**
Declares an exception. For the syntax of `exception_declaration`, see Exception Handler on page 13-45.

**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 Handler on page 13-45.

**function_declaration**
Declares a function. See Function Declaration and Definition on page 13-75.

**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 2–28, "Block with Multiple and Duplicate Labels" on page 2-25.

**name**
Is the label name (without the delimiters << and >>).

**object_declaration**
Declares an instance of an object type. To create an object type, use the CREATE TYPE Statement on page 14-65.

**object_ref_declaration**

**procedureDeclaration**
Declare a procedure. See Procedure Declaration and Definition on page 13-101.

**recordDeclaration**
Declares a user-defined record. For the syntax of `recordDeclaration`, see Record Definition on page 13-104.

**statement**
An executable (not declarative) statement. 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 data type specifier such as CHAR, DATE, or RECORD.

**variable_declaration**
Declares a variable. For the syntax of variable_declaration, see Constant on page 13-32.

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 1–1, "PL/SQL Block Structure"
- Example 1–5, "Assigning Values to Variables as Parameters of a Subprogram" on page 1-8
- Example 2–28, "Block with Multiple and Duplicate Labels" on page 2-25

**Related Topics**
- Comment on page 13-31
- Constant on page 13-32
- Exception Handler on page 13-45
- Function Declaration and Definition on page 13-75
- Procedure Declaration and Definition on page 13-101
- PL/SQL Blocks on page 1-4
CASE Statement

The CASE statement chooses from a sequence of conditions, and execute a corresponding statement.

The simple CASE statement evaluates a single expression and compares it to several potential values.

The searched CASE statement evaluates multiple Boolean expressions and chooses the first one whose value is TRUE.

The CASE statement is appropriate when a different action is to be taken for each alternative.

Syntax

**simple_case_statement ::=**

```plaintext
label_name CASE case_operand

WHEN when_operand THEN statement

ELSE statement;

END CASE label_name;
```

**searched_case_statement ::=**

```plaintext
label_name CASE

WHEN boolean_expression THEN statement

ELSE statement;

END CASE label_name;
```

*(statement ::= on page 13-12, boolean_expression ::= on page 13-58)*
Keyword and Parameter Descriptions

case_operand
An expression whose value is used to select one of several alternatives. Its value can be of any PL/SQL type except BLOB, BFILE, an object type, a PL/SQL record, an index-by table, a varray, or a nested table.

WHEN { when_operand | boolean_expression } THEN statement
The when_operands or boolean_expressions are evaluated sequentially. If the value of a when_operand equals the value of case_operand, or if the value of a boolean_expression is TRUE, the statement associated with that when_operand or boolean_expression executes, and the CASE statement ends. Subsequent when_operands or boolean_expressions are not evaluated.

The value of a when_operand can be of 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.

Caution: The statements can modify the database and invoke nondeterministic functions. There is no fall-through mechanism, as there is in the C switch statement.

ELSE statement [statement ]...
In the simple CASE statement, the statements execute if and only if no when_operand has the same value as case_operand.

In the searched CASE statement, the statements execute if and only if no boolean_expression has the value TRUE.

If you omit the ELSE clause, and there is no match (that is, no when_operand has the same value as case_operand, or no boolean_expression has the value TRUE), the system raises a CASE_NOT_FOUND exception.

Examples

- Example 1–10, "Using the IF-THEN-ELSE and CASE Statement for Conditional Control" on page 1-14
- Example 4–6, "Simple CASE Statement" on page 4-5
- Example 4–7, "Searched CASE Statement" on page 4-6

Related Topics

- Expression on page 13-57
- IF Statement on page 13-80
- CASE Expressions on page 2-40
- Testing Conditions (IF and CASE Statements) on page 4-2
- Using the Simple CASE Statement on page 4-5
- Using the Searched CASE Statement on page 4-6
See Also:

- *Oracle Database SQL Language Reference* for information about the NULLIF function
- *Oracle Database SQL Language Reference* for information about the COALESCE function
CLOSE Statement

The CLOSE statement closes a cursor or cursor variable, thereby allowing its resources to be reused.

After closing a cursor, you can reopen it with the OPEN statement. You must close a cursor before reopening it.

After closing a cursor variable, you can reopen it with the OPEN-FOR statement. You need not close a cursor variable before reopening it.

Syntax

close_statement ::=  

```
CLOSE  
cursor_name

cursor_variable_name

host_cursor_variable_name
```

Keyword and Parameter Descriptions

`cursor_name`
The name of an open explicit cursor that was declared within the current scope.

`cursor_variable_name`
The name of an open cursor variable that was declared in the current scope.

`host_cursor_variable_name`
The name of an open cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind argument.

Examples

- Example 4–24, "EXIT in a FOR LOOP" on page 4-19
- 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

- FETCH Statement on page 13-68
- OPEN Statement on page 13-94
- OPEN-FOR Statement on page 13-96
- Closing a Cursor on page 6-13
- Querying Data with PL/SQL on page 6-16
**Collection**

A collection groups elements of the same type in a specified order. Each element has a unique subscript that determines its position in the collection.

PL/SQL has three kinds of collections:

- Associative arrays (formerly called "PL/SQL tables" or "index-by tables")
- Nested tables
- Variable-size arrays (varrays)

Associative arrays can be indexed by either integers or strings. Nested tables and varrays are indexed by integers.

To create a collection, you first define a collection type, and then declare a variable of that type.

---

**Note:** This topic applies to collection types that you define inside a PL/SQL block or package, which are different from standalone stored collection types that you create with the `CREATE TYPE Statement` on page 14-65.

In a PL/SQL block or package, you can define all three collection types. With the `CREATE TYPE` statement, you can create nested table types and varray types, but not associative array types.

---

**Syntax**

```
collection_type_definition ::= 
```

```
TYPE type_name IS assoc_array_type_def nested_table_type_def varray_type_def;
```

**assoc_array_type_def ::=**

```
TABLE OF element_type NOT NULL INDEX BY PLS_INTEGER
```

*(element_type ::= on page 13-23)*

**nested_table_type_def ::=**

```
TABLE OF element_type NOT NULL INDEX BY BINARY_INTEGER
```

*(element_type ::= on page 13-23)*
**VARRAY**

```
VARRAY
VARYING
ARRAY
size_limit
OF
element_type
```

(element_type ::= on page 13-23)

**collection_variable_dec** ::= 
```
collection_name
type_name
```

**element_type** ::= 
```
cursor_name
% ROWTYPE

db_table_name
% ROWTYPE

object_name
% TYPE

REF

object_type_name

record_name
% TYPE

record_type_name

scalar_datatype_name

variable_name
% TYPE
```

**Keyword and Parameter Descriptions**

**collection_name**

The name that you give to the variable of the collection type that you defined.

**element_type**

The data type of the collection element (any PL/SQL data type except REF CURSOR).

For a nested table:

- If `element_type` is an object type, then the nested table type describes a table whose columns match the name and attributes of the object type.
- If `element_type` is a scalar type, then the nested table type describes a table with a single, scalar type column called COLUMN_VALUE.
- You cannot specify NCLOB for `element_type`. However, you can specify CLOB or BLOB.

**INDEX BY**

For an associative array, the data type of its indexes—PLS_INTEGER, BINARY_INTEGER, or VARCHAR2.
**NOT NULL**

Specifies that no element of the collection can have the value NULL.

**size_limit**

For a varray, a positive integer literal that specifies the maximum number of elements it can contain. A maximum limit is imposed. See Referencing Collection Elements on page 5-12.

**type_name**

The name that you give to the collection type that you are defining.

**v_size**

For an associative array, the length of the VARCHAR2 key by which it is indexed.

### Usage Notes

The type definition of an associative array can appear only in the declarative part of a block, subprogram, package specification, or package body.

The type definition of a nested table or varray can appear either in the declarative part of a block, subprogram, package specification, or package body (in which case it is local to the block, subprogram, or package) or in the CREATE TYPE Statement on page 14-65 (in which case it is a standalone stored type).

Nested tables extend the functionality of associative arrays, so they differ in several ways. See Choosing Between Nested Tables and Associative Arrays on page 5-6.

Nested tables and varrays can store instances of an object type and, conversely, can be attributes of an object type.

Collections work like the arrays of most third-generation programming languages. A collection has only one dimension. To model a multidimensional array, declare a collection whose items are other collections.

Collections can 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.

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.

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 about 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:

\[ \text{function}_\text{name}(\text{parameter}_\text{list})(\text{subscript}) \]

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 and Using an Associative Array" on page 5-2
- 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 Method Call on page 13-27
- CREATE TYPE Statement on page 14-65
- CREATE TYPE BODY Statement on page 14-83
- Record Definition on page 13-104
- Defining Collection Types on page 5-7
Collection Method Call

A collection method is a built-in PL/SQL subprogram that returns information about a collection or operates on a collection.

Syntax

\[
\textit{collection_method_call} ::= \\
\textit{collection_name} \\
\text{COUNT} \\
\text{DELETE} \\
\text{EXISTS} \\
\text{EXTEND} \\
\text{FIRST} \\
\text{LAST} \\
\text{LIMIT} \\
\text{NEXT} \\
\text{PRIOR} \\
\text{TRIM}
\]

Keyword and Parameter Descriptions

\textit{collection_name}

The name of a collection declared within the current scope.

\textbf{COUNT}

A function that returns the current number of elements in \textit{collection_name}.

\textbf{DELETE}

A procedure whose action depends on the number of indexes specified.

\texttt{DELETE} with no indexes specified deletes all elements from \textit{collection_name}.

\texttt{DELETE}(n) deletes the \texttt{n}th element from an associative array or nested table. If the \texttt{n}th element is null, \texttt{DELETE}(n) does nothing.

\texttt{DELETE}(m,n) deletes all elements in the range \texttt{m}..\texttt{n} from an associative array or nested table. If \texttt{m} is larger than \texttt{n} or if \texttt{m} or \texttt{n} is null, \texttt{DELETE}(m,n) does nothing.

\textbf{EXISTS}

A function that indicates whether the \texttt{index} is a valid element in \textit{collection_name}.

\textbf{EXTEND}

An extension of a collection to a specified size.

\textbf{FIRST}, \textbf{LAST}, \textbf{LIMIT}, \textbf{NEXT}, \textbf{PRIOR}, \textbf{TRIM}

Utility methods that manipulate the position and size of a collection.

See Also: Counting the Elements in a Collection (COUNT Method) on page 5-21
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.

**See Also:** Deleting Collection Elements (DELETE Method) on page 5-27

**EXISTS**
A function that returns TRUE if the `index`th element of `collection_name` exists; otherwise, it returns FALSE.

Typically, you use EXIST to avoid raising an exception when you reference a nonexistent element, and with `DELETE` to maintain sparse nested tables.

You cannot use EXIST if `collection_name` is an associative array.

**See Also:** Checking If a Collection Element Exists (EXISTS Method) on page 5-21

**EXTEND**
A procedure whose action depends on the number of indexes specified.

`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` if `collection_name` is an associative array.

**See Also:** Increasing the Size of a Collection (EXTEND Method) on page 5-25

**FIRST**
A function that returns the first (smallest) subscript or key value in a collection. If the collection is empty, FIRST returns NULL. If the collection contains only one element, FIRST and LAST return the same subscript value. If the collection is a varray, FIRST always returns 1.

For a collection indexed by integers, FIRST and LAST return the first and last (smallest and largest) index numbers.

For an associative array indexed by strings, FIRST and LAST return the lowest and highest key values. If the NLS_COMP initialization parameter is set to ANSI, the order is based on the sort order specified by the NLS_SORT initialization parameter.

**See Also:** Finding the First or Last Collection Element (FIRST and LAST Methods) on page 5-23

**index**
A numeric expression whose value has data type `PLS_INTEGER` or a data type implicitly convertible to `PLS_INTEGER` (see Table 3-10, "Possible Implicit PL/SQL Data Type Conversions" on page 3-31).
LAST
A function that returns the last (largest) subscript value in a collection. If the collection is empty, LAST returns NULL. If the collection contains only one element, FIRST and LAST return the same subscript value. For varrays, 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.

See Also: Finding the First or Last Collection Element (FIRST and LAST Methods) on page 5-23

LIMIT
A function that returns the maximum number of elements that collection_name can have. If collection_name has no maximum size, LIMIT returns NULL.

See Also: Checking the Maximum Size of a Collection (LIMIT Method) on page 5-22

NEXT
A function that returns the subscript that succeeds index n. If n has no successor, NEXT(n) returns NULL.

See Also: Looping Through Collection Elements (PRIOR and NEXT Methods) on page 5-24

PRIOR
A function that returns the subscript that precedes index n in a collection. If n has no predecessor, PRIOR(n) returns NULL.

See Also: Looping Through Collection Elements (PRIOR and NEXT Methods) on page 5-24

TRIM
A procedure.
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. TRIM operates on the internal size of a collection. If TRIM encounters deleted elements, it includes them in its tally.

You cannot use TRIM if is collection_name is an associative array.

See Also: Decreasing the Size of a Collection (TRIM Method) on page 5-26

Usage Notes
A collection method call can appear wherever a PL/SQL subprogram invocation can appear in a PL/SQL statement (but not in a SQL statement).

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.

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 5-22
- 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 5-28

**Related Topics**

- **Collection** on page 13-22
- **Using Collection Methods** on page 5-20
Comment

A comment is text that the PL/SQL compiler ignores. Its primary purpose is to document code, but you can also disable obsolete or unfinished pieces of code by turning them into comments. PL/SQL has both single-line and multiline comments.

Syntax

```
comment ::= 
```

Keyword and Parameter Descriptions

```
--
Turns the rest of the line into a single-line comment. Any text that wraps to the next line is not part of the comment.

/
Begins a comment, which can span multiple lines.

*/
Ends a comment.

text
Any text.
```

Usage Notes

A single-line comment can appear within a statement, at the end of a line. A single-line comment can appear inside a multiline comment.

---

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

---

A multiline comment can appear anywhere except within another multiline comment.

Examples

- Example 2–4, "Single-Line Comments" on page 2-9
- Example 2–5, "Multiline Comment" on page 2-10
**Constant**

A constant holds a value that does not change.

A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

### Syntax

- `declaration ::=`

  ```plaintext
  constant_name CONSTANT datatype NOT NULL DEFAULT expression
  ```

  *(expression ::= on page 13-57)*

### datatype ::=  

- `collection_name % TYPE`
- `collection_type_name % TYPE`
- `cursor_name % ROWTYPE`
- `cursor_variable_name % TYPE`
- `db_table_name % ROWTYPE`
- `column_name % TYPE`
- `object_name % TYPE`
- `object_type_name % REF object_type_name`
- `record_name % TYPE`
- `record_type_name % REF record_type_name`
- `ref_cursor_type_name % TYPE`
- `scalar_datatype_name % TYPE`
- `variable_name % TYPE`

### Keyword and Parameter Descriptions

- **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 data type specifier TABLE or VARRAY.

- **constant_name**
  The name of the constant. For naming conventions, see Identifiers on page 2-4.
**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**
The value to be assigned to the constant when the declaration is elaborated. The value of expression must be of a data type that is compatible with the data type of the constant.

**NOT NULL**
A constraint that prevents the program from assigning a null value to the constant. Assigning a null to a variable defined as NOT NULL raises the predefined exception VALUE_ERROR.

**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 data type specifier RECORD.

**ref_cursor_type_name**
A user-defined cursor variable type, defined using the data type 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 data types as columns in the row.

**scalar_datatype_name**
A predefined scalar data type such as BOOLEAN, NUMBER, or VARCHAR2. Includes any qualifiers for size, precision, and character or byte semantics.
**%TYPE**

Represents the data type of a previously declared collection, cursor variable, field, object, record, database column, or variable.

**Usage Notes**

Constants are initialized every time a block or subprogram is entered. Whether public or private, constants declared in a package specification are initialized only once for each session.

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 2-7, "Declaring Constants" on page 2-11

**Related Topics**

- Collection on page 13-22
- Variable on page 13-130
- Comments on page 2-9
- Constants on page 2-11
CONTINUE Statement

The CONTINUE statement exits the current iteration of a loop, either conditionally or unconditionally, and transfer control to the next iteration. You can name the loop to be exited.

Syntax

```
continue_statement ::= 

CONTINUE label_name WHEN boolean_expression ;
```

(boolean_expression ::= on page 13-58)

Keyword and Parameter Descriptions

**boolean_expression**
If and only if the value of this expression is TRUE, the current iteration of the loop (or the iteration of the loop identified by label_name) is exited immediately.

**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

A CONTINUE statement can appear anywhere inside a loop, but not outside a loop.

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 4-10
- Example, "Using the CONTINUE-WHEN Statement" on page 4-11

Related Topics

- EXIT Statement on page 13-51
- Expression on page 13-57
- LOOP Statements on page 13-88
- Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements) on page 4-8
Cursor Attribute

Every explicit cursor and cursor variable has four attributes, each of which returns useful information about the execution of a data manipulation statement.

Syntax

cursor_attribute ::= 

cursor_name

cursor_variable_name

host_cursor_variable_name

%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 data type 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 Variable Declaration on page 13-39
- Explicit Cursor on page 13-53
- SQL (Implicit) Cursor Attribute on page 13-122
- Attributes of Explicit Cursors on page 6-13
Cursor Variable Declaration

A cursor variable points to the unnamed work area in which the database stores processing information when it executes a multiple-row query. With this pointer to the work area, you can access its information, and process the rows of the query individually.

A cursor variable is like a C or Pascal pointer, which holds the address of an item instead of the item itself.

To create a cursor variable, define a `REF CURSOR` type, and then declare the cursor variable to be of that type. Declaring a cursor variable creates a pointer, not an item.

Syntax

```
ref_cursor_type_definition ::= 
  TYPE type_name IS REF CURSOR

cursor_variable_declaration ::= 
  cursor_variable_name type_name
```

Keyword and Parameter Descriptions

- **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 data type specifier `RECORD`. 
**REF CURSOR**

Cursor variables all have the data type REF CURSOR.

**RETURN**

Specifies the data type 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 data type 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 data types.

**%TYPE**

Provides the data type of a previously declared user-defined record.

**type_name**

A user-defined cursor variable type that was defined as a REF CURSOR.

**Usage Notes**

A REF CURSOR type definition can appear either in the declarative part of a block, subprogram, package specification, or package body (in which case it is local to the block, subprogram, or package) or in the CREATE TYPE Statement on page 14-65 (in which case it is a standalone stored type).

A cursor variable declaration can appear only in the declarative part of a block, subprogram, or package body (not in a package specification).

Cursor variables are 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 data type 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 data type. However, if one or both cursor variables are weakly typed, they need not have the same data type.

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 13-20
- Cursor Attribute on page 13-37
- Explicit Cursor on page 13-53
- FETCH Statement on page 13-68
- OPEN-FOR Statement on page 13-96
- Using Cursor Variables (REF CURSORs) on page 6-22
- Declaring REF CURSOR Types and Cursor Variables on page 6-23
EXCEPTION_INIT Pragma

The EXCEPTION_INIT pragma associates a user-defined exception name with an Oracle Database error number. You can intercept any Oracle Database error number and write an exception handler for it, instead of using the OTHERS handler.

Syntax

\[
\text{exception_initPragma} ::= \\
\text{PRAGMA EXCEPTION_INIT} \text{exception_name} \text{error_number}
\]

Keyword and Parameter Descriptions

**error_number**
Any valid Oracle Database 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.
Be sure to assign only one exception name to an error number.

**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

A EXCEPTION_INIT pragma can appear only in the same declarative part as its associated exception, anywhere after the exception declaration.

Examples

- Example 11–4, "Using PRAGMA EXCEPTION_INIT" on page 11-8
- Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-16

Related Topics

- Exception Declaration on page 13-44
- Exception Handler on page 13-45
- SQLCODE Function on page 13-125
- SQLERRM Function on page 13-126
- Associating a PL/SQL Exception with a Number (EXCEPTION_INIT Pragma) on page 11-7
Exception Declaration

An exception declaration declares a user-defined exception.

Unlike a predefined exception, a user-defined exception must be raised explicitly with either a RAISE statement or the procedure DBMS_STANDARD_RAISE_APPLICATION_ERROR. The latter lets you associate an error message with the user-defined exception.

Syntax

\[
\text{exception_declaration ::=}
\]

 Keyword and Parameter Descriptions

**exception_name**

The name you give to the user-defined exception.

---

**Caution:** Using the name of a predefined exception for `exception_name` is not recommended. For details, see Redeclaring Predefined Exceptions on page 11-9.

---

Example

Example 1–12, "Using WHILE-LOOP for Control" on page 1-16
Example 1–16, "Creating a Standalone PL/SQL Procedure" on page 1-18
Example 2–28, "Block with Multiple and Duplicate Labels" on page 2-25
Example 5–35, "Using TRIM to Decrease the Size of a Collection" on page 5-26
Example 5–38, "Collection Exceptions" on page 5-29
Example 6–37, "Using ROLLBACK" on page 6-34
Example 7–13, "Using Validation Checks to Guard Against SQL Injection" on page 7-16
Example 8–1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3
Example 10–3, "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 11-7
Example 11–9, "Reraising a PL/SQL Exception" on page 11-13
Example 12–6, "Using Rollbacks with FORALL" on page 12-14
Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-16

Related Topics

- Exception Handler on page 13-45
- RAISE Statement on page 13-103
- Defining Your Own PL/SQL Exceptions on page 11-6
- Declaring PL/SQL Exceptions on page 11-6
Exception Handler

An exception handler processes a raised exception (run-time error or warning condition). The exception can be either predefined or user-defined. Predefined exceptions are raised implicitly (automatically) by the run-time system. must be raised explicitly with either a RAISE statement or the procedure DBMS_STANDARD_RAISE_APPLICATION_ERROR. The latter lets you associate an error message with the user-defined exception.

Syntax

```
exception_handler ::= 

    (WHEN exception_name OR OTHERS THEN statement)
```

*(statement ::= on page 13-12)*

Keyword and Parameter Descriptions

**exception_name**
The name of either 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.

**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 Table 11–1 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 1–12, "Using WHILE-LOOP for Control" on page 1-16
Example 1–16, "Creating a Standalone PL/SQL Procedure" on page 1-18
Example 2–28, "Block with Multiple and Duplicate Labels" on page 2-25
Example 5–35, "Using TRIM to Decrease the Size of a Collection" on page 5-26
Example 5–38, "Collection Exceptions" on page 5-29
Example 6–37, "Using ROLLBACK" on page 6-34
Example 7–13, "Using Validation Checks to Guard Against SQL Injection" on page 7-16
Example 8–1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3
Example 10–3, "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 11-7
Example 11–9, "Reraising a PL/SQL Exception" on page 11-13
Example 12–6, "Using Rollbacks with FORALL" on page 12-14
Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-16

**Related Topics**

- **Block** on page 13-8
- **EXCEPTION_INIT Pragma** on page 13-43
- **Exception Declaration** on page 13-44
- **RAISE Statement** on page 13-103
- **SQLCODE Function** on page 13-125
- **SQLERRM Function** on page 13-126
- **Handling Raised PL/SQL Exceptions** on page 11-13
EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement builds and executes a dynamic SQL statement in a single operation. It is the means by which native dynamic SQL processes most dynamic SQL statements.

Syntax

`execute_immediate_statement ::=`

```
EXECUTE IMMEDIATE dynamic_sql_stmt
```  

`into_clause ::=`

```
INTO variable_name, variable_name
```  

`bulk_collect_into_clause ::=`

```
BULK COLLECT INTO collection_name:host_array_name
```  

`using_clause ::=`

```
USING IN OUT IN OUT bind_argument
```  

Keyword and Parameter Descriptions

`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 13-111.

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`. 

---

EXECUTE IMMEDIATE Statement

PL/SQL Language Elements  13-43
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 a of 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 about 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" on page 7-3
- Example 7–2, "Unsupported Data Type in Native Dynamic SQL" on page 7-3
- Example 7–3, "Uninitialized Variable for NULL in USING Clause" on page 7-4
- Example 7–5, "Repeated Placeholder Names in Dynamic PL/SQL Block" on page 7-6

Related Topics

- OPEN-FOR Statement on page 13-96
- RETURNING INTO Clause on page 13-111
- Using the EXECUTE IMMEDIATE Statement on page 7-2
- Using DBMS_SQL Package on page 7-6
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.

Syntax

exit_statement ::= 

( boolean_expression ::= on page 13-58)

Keyword and Parameter Descriptions

boolean_expression
If and only if the value of this expression is TRUE, the current loop (or the loop labeled by label_name) is exited immediately.

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

An EXIT statement can appear anywhere inside a loop, but not outside a loop. 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 4–9, "EXIT Statement" on page 4-9
- Example 4–24, "EXIT in a FOR LOOP" on page 4-19
- Example 4–25, "EXIT with a Label in a FOR LOOP" on page 4-19

Related Topics

- CONTINUE Statement on page 13-35
- Expression on page 13-57
- LOOP Statements on page 13-88
- Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements) on page 4-8
Explicit Cursor

An explicit cursor names the unnamed work area in which the database stores processing information when it executes a multiple-row query. When you have named the work area, you can access its information, and process the rows of the query individually.

Syntax

```plsql
cursor_declaration ::= 

CURSOR cursor_name (

   cursor_parameter_declaration

) RETURN rowtype IS

select_statement;
```

(cursor_parameter_declaration ::= on page 13-54, rowtype ::= on page 13-54)

```plsql
cursor_spec ::= 

CURSOR cursor_name (

   cursor_parameter_declaration

) RETURN rowtype;
```

(cursor_parameter_declaration ::= on page 13-54, rowtype ::= on page 13-54)

```plsql
cursor_body ::= 

CURSOR cursor_name (

   cursor_parameter_declaration

) RETURN rowtype IS

select_statement;
```

(cursor_parameter_declaration ::= on page 13-54, rowtype ::= on page 13-54)

```plsql
cursor_parameter_declaration ::= 

parameter_name IN datatype := DEFAULT expression
```

(expression ::= on page 13-57)
Explicit Cursor

\[ \text{rowtype} ::= \]

\[ \begin{align*}
&\text{db_table_name} \quad \% \rightarrow \text{ROWTYPE} \\
&\text{cursor_name} \quad \% \rightarrow \text{ROWTYPE} \\
&\text{cursor_variable_name} \quad \% \rightarrow \text{ROWTYPE} \\
&\text{record_name} \quad \% \rightarrow \text{TYPE} \\
&\text{record_type_name} \quad \% \rightarrow \text{TYPE}
\end{align*} \]

Keyword and Parameter Descriptions

**cursor_name**
An explicit cursor previously declared within the current scope.

**datatype**
A type specifier. For the syntax of *datatype*, see Constant on page 13-32.

**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 data types.

---

**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 data type specifier RECORD.

**RETURN**
Specifies the data type 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 data type of a previously declared record.

A cursor body must have a SELECT statement and the same RETURN clause as its corresponding cursor specification. Also, the number, order, and data types 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 data types.

select_statement
A SQL SELECT statement. If the cursor declaration declares parameters, each parameter must appear in select_statement.

See: Oracle Database SQL Language Reference for SELECT statement syntax

%TYPE
Provides the data type of a previously declared user-defined record.

Usage Notes
You must declare a cursor before referencing it in an OPEN, FETCH, or CLOSE statement.

Note: An explicit cursor declared in a package specification is affected by the AUTHID clause of the package. For more information, see "CREATE PACKAGE Statement" on page 14-39.

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 SQL 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 2-22.

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 data type 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
Explicit Cursor

- Example 6–30, "Stored Procedure to Open Ref Cursors with Different Queries" on page 6-26

Related Topics
- CLOSE Statement on page 13-20
- Cursor Attribute on page 13-37
- Cursor Variable Declaration on page 13-39
- FETCH Statement on page 13-68
- OPEN Statement on page 13-94
- SELECT INTO Statement on page 13-116
- Declaring a Cursor on page 6-10
- Querying Data with PL/SQL on page 6-16
Expression

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function calls, and placeholders) and operators. The simplest expression is a single variable.

The PL/SQL compiler determines the data type of an expression from the types of the operands and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results.

Syntax

\[
\text{expression ::= }
\]

\[
\text{boolean_expression ::= }
\]

\[
\text{AND } \quad \text{OR } \quad \text{NOT }
\]

\[
\text{boolean_constant_name } \\
\text{boolean_function_call } \\
\text{boolean_literal } \\
\text{boolean_variable_name } \\
\text{other_boolean_form }
\]

\[
(\text{expression})
\]
other_boolean_form ::= 

character_expression ::= 

13-52 Oracle Database PL/SQL Language Reference
numeric_subexpression ::= 

cursor_name 
cursor_variable_name
::=

SQL

cursor_name

cursor_variable_name

host_cursor_variable_name

% ROWCOUNT

SQL

% BULK_ROWCOUNT

to

integer

indicator_name


numeric_constant_name

numeric_function_call

numeric_literal

numeric_variable_name

collection_name

COUNT

FIRST

LAST

LIMIT

NEXT

PRIOR

(index)

**

exponent

numeric_expression

+ 

–


numeric_subexpression

date_expression ::= 

date_constant_name

date_function_call

date_literal

date_variable_name

host_variable_name

indicator_name


+ 

–

numeric_expression

numeric_expression ::= 

numeric_subexpression

+ 

–


numeric_subexpression
**simple_case_expression ::=**

```
CASE case_operand
  WHEN case_operand_value THEN result_value
  ELSE result_value
END CASE;
```

**searched_case_expression ::=**

```
CASE
  WHEN boolean_expression THEN result_value
  ELSE result_value
END CASE;
```

*(boolean_expression ::= on page 13-58)*

**Keyword and Parameter Descriptions**

**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 whose value is Boolean (TRUE, FALSE, or NULL).

**boolean_function_call**
A call to a function 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 (Implicit) Cursor Attribute on page 13-122.

**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 Method Call](#) on page 13-27.

**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 data type of the host variable must be implicitly convertible to the appropriate PL/SQL data type. 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 data type.

**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 2-30. 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 2-30.

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 2-34.

SQL
A cursor opened implicitly by the database 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.

case_operand
An expression whose value is used to select one of several alternative result values. The value of case_operand can be of any PL/SQL type except BLOB, BFILE, an object type, a PL/SQL record, an index-by table, a varray, or a nested table.

WHEN { case_operand_value | boolean_expression } THEN result_value
The case_operand_values or boolean_expressions are evaluated sequentially. If a case_operand_value is the value of case_operand, or if the value of a boolean_expression is TRUE, the result_value associated with that case_operand_value or boolean_expression is returned. Subsequent case Operand_values or boolean_expressions are not evaluated.

A case_operand_value can be of 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.

ELSE result_value
In the simple CASE expression, the result_value is returned if and only if no case_operand_value has the same value as case_operand.

In the searched CASE statement, the result_value is returned if and only if no boolean_expression has the value TRUE.

If you omit the ELSE clause, the case expression returns NULL.

Usage Notes

In a Boolean expression, you can only compare values that have compatible data types. For more information, see PL/SQL Data Type Conversion on page 3-28.

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, data type 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, \texttt{NOT} has the highest precedence, \texttt{AND} has the next-highest precedence, and \texttt{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

- Example 1–3, "Assigning Values to Variables with the Assignment Operator" on page 1-7
- Using the WHEN Clause with a CASE Statement on page 2-41
- Using a Search Condition with a CASE Statement on page 2-41

Related Topics

- Assignment Statement on page 13-3
- CASE Statement on page 13-17
- Constant on page 13-32
- EXIT Statement on page 13-51
- IF Statement on page 13-80
- LOOP Statements on page 13-88
- PL/SQL Expressions and Comparisons on page 2-28
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.

Syntax

fetch_statement ::= 

into_clause ::= 

bulk_collect_into_clause ::= 

Keyword and Parameter Descriptions

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_cursor_variable_name**

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The data type of the host cursor variable is compatible with the return type of any PL/SQL cursor variable.

**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 data type conversions are needed, multiple composite targets cannot be used in the BULK COLLECT INTO clause.
- When an implicit data type 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 6-31
- 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 13-20
- Cursor Variable Declaration on page 13-39
- Explicit Cursor on page 13-53
- LOOP Statements on page 13-88
- OPEN Statement on page 13-94
- OPEN-FOR Statement on page 13-96
- RETURNING INTO Clause on page 13-111
- Querying Data with PL/SQL on page 6-16
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.

Syntax

forall_statement ::= …

bounds_clause ::= ...

Keyword and Parameter Descriptions

**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 12-16.

### 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 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.

- Collection subscripts must be just the index variable rather than an expression,
such as i rather than i+1.

- The cursor attribute %BULK_ROW_COUNT 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,
ot 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 12-10
- Example 12–3, "Issuing INSERT Statements in a Loop" on page 12-11
- Example 12–4, "Using FORALL with Part of a Collection" on page 12-11
- Example 12–5, "Using FORALL with Nonconsecutive Index Values" on page 12-12
- Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-16
- Example 12–16, "Using FORALL with BULK COLLECT" on page 12-21

Related Topics

- Reducing Loop Overhead for DML Statements and Queries with Bulk SQL on
page 12-9
- Retrieving Query Results into Collections (BULK COLLECT Clause) on page 12-17
Function Declaration and Definition

A function is a subprogram that returns a single value. You must declare and define a function before invoking it. You can either declare and define it at the same time, or you can declare it first and then define it later in the same block.

**Note:** This topic applies to functions that you declare and define inside a PL/SQL block or package, which are different from standalone stored functions that you create with the `CREATE FUNCTION Statement` on page 14-29.

**Syntax**

```
function_declaration ::= 

  function_heading ::= 

(function_heading ::= on page 13-99, datatype ::= on page 13-32)

function_definition ::= 

  (body ::= on page 13-11, declare_section ::= on page 13-8)
```
result_cache_clause ::= 

Keyword and Parameter Descriptions

**body**
The required executable part of the function and, optionally, the exception-handling part of the function.

At least one execution path must lead to a `RETURN` statement; otherwise, you get a run-time error.

**data_source**
The name of either a database table or a database view.

**declare_section**
The optional declarative part of the function. Declarations are local to the function, can be referenced in `body`, and cease to exist when the function completes execution.

**DETERMINISTIC**
Specify `DETERMINISTIC` to indicate that the function returns the same result value whenever it is invoked with the same values for its parameters. This helps the optimizer avoid redundant function calls: If a stored function was invoked previously with the same arguments, the optimizer can elect to use the previous result.

Do not specify `DETERMINISTIC` for a function whose result depends on the state of session variables or schema objects, because results might vary across calls. Instead, consider making the function result-cached (see Making Result-Cached Functions Handle Session-Specific Settings on page 8-33 and Making Result-Cached Functions Handle Session-Specific Application Contexts on page 8-34).

Only `DETERMINISTIC` functions can be invoked from a function-based index or a materialized view that has query-rewrite enabled. For more information and possible limitations of the `DETERMINISTIC` option, see CREATE FUNCTION Statement on page 14-29.

See Also: CREATE INDEX statement in Oracle Database SQL Language Reference

**function_declaration**
Declares a function, but does not define it. The definition must appear later in the same block or subprogram as the declaration.

A function declaration is also called a function specification, or function spec.

**function_definition**
Either defines a function that was declared earlier in the same block or subprogram, or declares and defines a function.
**function_name**
The name that you give to the function that you are declaring or defining.

**IN, OUT, IN OUT**
Parameter modes that define the action of formal parameters. For summary information about parameter modes, see Table 8–1 on page 8-9.

**NOCOPY**
Specify NOCOPY to instruct the database to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an index-by table, or a varray to an OUT or IN OUT parameter. IN parameter values are always passed NOCOPY.

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter, or assignments made to this parameter may show immediately in a package variable, if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the function is exited with an unhandled exception, then any assignment made to this parameter may be visible in the caller’s variable.

These effects may or may not occur on any particular call. You should use NOCOPY only when these effects would not matter.

**parameter_name**
The name of the formal parameter that you are declaring, which you can reference in body.

**PIPELINED**
PIPELINED specifies to return the results of a table function iteratively. A table function returns a collection type (a nested table or varray) with elements that are SQL data types. You can query table functions using the TABLE keyword before the function name in the FROM clause of a SQL query. For more information, see Performing Multiple Transformations with Pipelined Table Functions on page 12-34.

**RELIES_ON**
Specifies the data sources on which the results of a function depend. For more information, see Using the PL/SQL Function Result Cache on page 8-27.

**RESULT_CACHE**
Causes the results of the function to be cached. For more information, see Using the PL/SQL Function Result Cache on page 8-27.

**RETURN datatype**
For datatype, specify the data type of the return value of the function. The return value can have any data type supported by PL/SQL. You cannot constrain this data type (with NOT NULL, for example).

{ := | DEFAULT } expression
Specifies a default value for an IN parameter. If the invoker of the function specifies a value for the parameter, then expression is not evaluated for that invocation (see
Example 8–7). Otherwise, the parameter is initialized to the value of \textit{expression}. The value and the parameter must have compatible data types.

**Examples**

- Example 8–2, "Declaring, Defining, and Invoking a Simple PL/SQL Function" on page 8-5
- Example 5–44, "Returning a Record from a Function" on page 5-33

**Related Topics**

- Parameter Declaration on page 13-99
- Procedure Declaration and Definition on page 13-101
- Using the PL/SQL Function Result Cache on page 8-27
- Chapter 8, "Using PL/SQL Subprograms"

\textbf{See Also:} \textit{Oracle Database Advanced Application Developer's Guide} for information about restrictions on user-defined functions that are called from SQL statements and expressions
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 labeled statement or block.

**Syntax**

```
label_declaration ::= 

  << label_name >>

goto_statement ::= 

  GOTO << label_name >>;
```

**Keyword and Parameter Descriptions**

- **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 4–26, "Simple GOTO Statement" on page 4-20
- Example 4–29, "Using a GOTO Statement to Branch to an Enclosing Block" on page 4-22
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

\( \text{if}\_\text{statement} \::= \)

- IF boolean_expression THEN statement
- ELSIF boolean_expression THEN statement
- ELSE statement END IF

(boolean_expression ::= on page 13-58)

Keyword and Parameter Descriptions

**boolean_expression**
If and only if the value of this expression is TRUE, the statements following THEN execute.

**ELSE**
If control reaches this keyword, the statements that follow it execute. This occurs when no boolean_expression had the value TRUE.

**ELSIF**
Introduces a boolean_expression that is evaluated if no preceding boolean_expression had the value 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 1–10, "Using the IF-THEN-ELSE and CASE Statement for Conditional Control" on page 1-14
- Example 4–1, "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-THEN-ELSE Statements" on page 4-3
- Example 4–4, "Using the IF-THEN-ELSIF Statement" on page 4-4

Related Topics

- CASE Statement on page 13-17
- Expression on page 13-57
- Testing Conditions (IF and CASE Statements) on page 4-2
- Using the GOTO Statement on page 4-20
**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.

**Syntax**

```plaintext
inlinePragma ::= 
```

**Keyword and Parameter Descriptions**

**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.

**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
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
...

PRAGMA INLINE ... 'NO' overrides PRAGMA INLINE ... 'YES' for the same subprogram, regardless of their order in the code. In Example 13–4, the second INLINE pragma overrides both the first and third INLINE pragmas.

Example 13–4  Applying Two INLINE Pragmas to the Same Subprogram

PROCEDURE p1 (x PLS_INTEGER) IS ...

...

PRAGMA INLINE (p1, 'YES');
PRAGMA INLINE (p1, 'NO');
PRAGMA INLINE (p1, 'YES');

x := p1(1) + p1(2) + 17;   -- These 2 calls to p1 will not be inlined
...

Related Topics

■ How PL/SQL Optimizes Your Programs on page 12-1
Literal

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.

Syntax

\[
\begin{align*}
\text{numeric_literal} & ::= \\
\text{integer_literal} & ::= \\
\text{real_number_literal} & ::= \\
\text{character_literal} & ::= \\
\text{string_literal} & ::= \\
\text{booleanLiteral} & ::= 
\end{align*}
\]

\text{integer} \quad \text{real} \quad \text{character} \quad \text{string} \quad \text{boolean}
Keyword and Parameter Descriptions

**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 more information, see Numeric Literals on page 2-6.

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 more information, see Character Literals on page 2-7.

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 more information, including NCHAR string literals, see String Literals on page 2-7.

The BOOLEAN values TRUE and FALSE cannot be inserted into a database column. For more information, see BOOLEAN Literals on page 2-8.

**Examples**

- **Numeric literals:**
  
  25 6.34 7E2 25e-03 .1 1. +17 -4.4 -4.5D -4.6F

- **Character literals:**
  
  'H'  '6'  '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.#'
Literal

- Example 2-3, "Using DateTime Literals" on page 2-8
- Example 2-56, "Using Conditional Compilation with Database Versions" on page 2-54

Related Topics
- Constant on page 13-32
- Expression on page 13-57
- Literals on page 2-6
Loop Statements

A LOOP statement executes a sequence of statements multiple times. PL/SQL provides these loop statements:

- Basic loop
- WHILE loop
- FOR loop
- Cursor FOR loop

Syntax

**basic_loop_statement ::=**

```
< label_name >= LOOP statement END LOOP < label_name >
```

**while_loop_statement ::=**

```
< label_name >= WHILE boolean_expression LOOP statement END LOOP < label_name >
```

( boolean_expression ::= on page 13-58)

**for_loop_statement ::=**

```
< label_name >= FOR index_name IN < REVERSE lower_bound .. upper_bound > LOOP statement END LOOP < label_name >
```

LOOP Statements

**cursor_for_loop_statement** ::=  

```
<< label_name >> FOR record_name IN 
```

```
cursor_name 
```

```
( cursor_parameter_name )
```

```
( select_statement )
```

```
LOOP statement END LOOP label_name
```

**Keyword and Parameter Descriptions**

**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**

If and only if the value of this expression is `TRUE`, the statements after `LOOP` execute.

**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 Explicit Cursor on page 13-53. 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 4-22, "Referencing Global Variable with Same Name as Loop Counter" on page 4-18.

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 labeled 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 4-23, "Referencing Outer Counter with Same Name as Inner Counter" on page 4-18.

**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:

```plsql
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 Data Types 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 data types as their corresponding columns. To access field values, you use dot notation, as follows:

```plaintext
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`:

```sql
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:

```sql
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 13-116. 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 4–25, "EXIT with a Label in a FOR LOOP" on page 4-19
- 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

- CONTINUE Statement on page 13-35
- EXIT Statement on page 13-51
- Explicit Cursor on page 13-53
- FETCH Statement on page 13-68
- FORALL Statement on page 13-72
- OPEN Statement on page 13-94
- Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements) on page 4-8
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.

**Syntax**

```null
null_statement ::= 
```

**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 1–16, "Creating a Standalone PL/SQL Procedure" on page 1-18
- Example 1–8, "Declaring a Record Type" on page 1-12
- Example 4–28, "Using a NULL Statement to Allow a GOTO to a Label" on page 4-21
- Example 4–31, "Using the NULL Statement to Show No Action" on page 4-23
- Example 4–32, "Using NULL as a Placeholder When Creating a Subprogram" on page 4-24

**Related Topics**

- Sequential Control (GOTO and NULL Statements) on page 4-20
- Using the NULL Statement on page 4-23
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.

Syntax

open_statement ::=  

Keyword and Parameter Descriptions

**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 Explicit Cursor on page 13-53.) 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 data types 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 13-20
- Explicit Cursor on page 13-53
- FETCH Statement on page 13-68
- LOOP Statements on page 13-88
- Querying Data with PL/SQL on page 6-16
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.

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.

Syntax

open_for_statement ::=  

using_clause ::=  

Keyword and Parameter Descriptions

**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 data type 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).

**dynamic_string**

A string literal, string variable, or string expression that represents any SQL statement. It must be of type CHAR, VARCHAR2, or CLOB.
OPEN-FOR Statement

USING
Used only if select_statement includes placeholders, this clause specifies a list of bind arguments.

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.

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:

```sql
/* 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 need not 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" on page 7-4

Related Topics
- CLOSE Statement on page 13-20
- **Cursor Variable Declaration** on page 13-39
- **EXECUTE IMMEDIATE Statement** on page 13-47
- **FETCH Statement** on page 13-68
- **LOOP Statements** on page 13-88
- **Using Cursor Variables (REF CURSORs)** on page 6-22
- **Using the OPEN-FOR, FETCH, and CLOSE Statements** on page 7-4
Parameter Declaration

A parameter declaration can appear in the following:

- Function Declaration and Definition on page 13-75
- Procedure Declaration and Definition on page 13-101
- CREATE FUNCTION Statement on page 14-29
- CREATE PROCEDURE Statement on page 14-46

Syntax

`parameter_declaration ::=`

\[\text{parameter_name} \rightarrow \text{IN}, \text{OUT}, \text{IN OUT}, \text{NOCOPY}\]

\[\text{datatype} ::= \text{on page 13-32}, \text{expression} ::= \text{on page 13-57}\]

Keyword and Parameter Descriptions

`datatype`

The data type of the parameter that you are declaring. You cannot constrain this data type (with NOT NULL, for example).

`IN, OUT, IN OUT`

Parameter modes that define the action of formal parameters. For summary information about parameter modes, see Table 8–1 on page 8-9.

**Note:** Avoid using `OUT` and `IN OUT` with functions. The purpose of a function is to take zero or more parameters and return a single value. Functions must be free from side effects, which change the values of variables not local to the subprogram.

`NOCOPY`

Specify `NOCOPY` to instruct the database to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an index-by table, or a varray to an `OUT` or `IN OUT` parameter. `IN` parameter values are always passed `NOCOPY`.

- When you specify `NOCOPY`, assignments made to a package variable may show immediately in this parameter, or assignments made to this parameter may show immediately in a package variable, if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
If the subprogram is exited with an unhandled exception, then any assignment made to this parameter may be visible in the caller’s variable. These effects might or might not occur on any particular call. Use NOCOPY only when these effects would not matter.

**parameter_name**
The name of the formal parameter that you are declaring, which you can reference in the body of the subprogram.

```
{ := | DEFAULT } expression
```
Specifies a default value for an **IN** parameter. If the invoker of the subprogram specifies a value for the parameter, then *expression* is not evaluated for that invocation (see Example 8–7). Otherwise, the parameter is initialized to the value of *expression*. The value and the parameter must have compatible data types.

### Examples

- Example 8–5, “Using OUT Mode” on page 8-8
- Example 8–6, “Procedure with Default Parameter Values” on page 8-10
- Example 8–7, “Formal Parameter with Expression as Default Value” on page 8-10

### Related Topics

- Function Declaration and Definition on page 13-75
- Procedure Declaration and Definition on page 13-101
- CREATE FUNCTION Statement on page 14-29
- CREATE PROCEDURE Statement on page 14-46
- Declaring and Passing Subprogram Parameters on page 8-6
A **procedure** is a subprogram that performs a specific action.

You must declare and define a procedure before invoking it. You can either declare and define it at the same time, or you can declare it first and then define it later in the same block or subprogram.

---

**Note:** This topic applies to procedures that you declare and define inside a PL/SQL block or package, which are different from standalone stored procedures that you create with the CREATE PROCEDURE Statement on page 14-46.

---

**Syntax**

```
procedure_declaration ::=  
  procedure_heading

procedure_heading ::=  
  PROCEDURE procedure_name (parameter_declaration)  
    IS  
      AS  
      declare_section  
      body

(body := on page 13-11, declare_section := on page 13-8)
```

---

**Keyword and Parameter Descriptions**

**body**

The required executable part of the procedure and, optionally, the exception-handling part of the procedure.

**declare_section**

The optional declarative part of the procedure. Declarations are local to the procedure, can be referenced in `body`, and cease to exist when the procedure completes execution.

**procedure_declaration**

Declares a procedure, but does not define it. The definition must appear later in the same block or subprogram as the declaration.

A procedure declaration is also called a **procedure specification**, or **procedure spec**.
**procedure_definition**
Either defines a procedure that was declared earlier in the same block or subprogram, or declares and defines a procedure.

**procedure_name**
The name that you give to the procedure that you are declaring or defining.

**Examples**
- Example 1-15, "PL/SQL Procedure" on page 1-18
- Example 8-1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3

**Related Topics**
- Function Declaration and Definition on page 13-75
- Parameter Declaration on page 13-99
- CREATE PROCEDURE Statement on page 14-46
- Chapter 8, "Using PL/SQL Subprograms"
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.

Syntax

raise_statement ::= 

- RAISE exception_name

Keyword and Parameter Descriptions

exception_name

A predefined or user-defined exception. For a list of the predefined exceptions, see 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 1–16, "Creating a Standalone PL/SQL Procedure" on page 1-18
- Example 10–3, "Creating the emp_admin Package" on page 10-6
- Example 11–3, "Scope of PL/SQL Exceptions" on page 11-7
- Example 11–9, "Reraising a PL/SQL Exception" on page 11-13

Related Topics

- Exception Handler on page 13-45
- Defining Your Own PL/SQL Exceptions on page 11-6
Record Definition

A record is a composite variable that can store data values of different types, similar to a `struct` type in C, C++, or Java.

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 ::= TYPE type_name IS_RECORD (field_declaration)
```

```
record_field_declaration ::= field_name datatype NOT_NULL := DEFAULT expression
```

```
record_type_declaration ::= record_name type_name
```

Keyword and Parameter Descriptions

**datatype**
A data type specifier. For the syntax of `datatype`, see Constant on page 13-32.

**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 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 data types.

**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 data type specifier RECORD.

:= I 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 data type 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 data type. (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 data types.
- 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 specification. 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 1–8, "Declaring a Record Type" on page 1-12
- Example 5–8, "VARRAY of Records" on page 5-10
- Example 5–20, "Assigning Values to VARRAYs with Complex Data Types" on page 5-16
■ Example 5–21, "Assigning Values to Tables with Complex Data Types" on page 5-16
■ 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 5-34
■ 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 on page 13-22
■ Function Declaration and Definition on page 13-75
■ Procedure Declaration and Definition on page 13-101
■ Defining and Declaring Records on page 5-31
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 13-75) instead of RESTRICT_REFERENCES.

The RESTRICT_REFERENCES pragma asserts that a user-defined subprogram does not read or write database tables or package variables.

Subprograms that read or write database tables or package variables are difficult to optimize, because any call to the subprogram might produce different results or encounter errors.

Syntax

\[
\text{restrict\_references\_pragma ::=}
\]

Keyword and Parameter Descriptions

**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.

**subprogram\_name**
The name of a user-defined subprogram, usually a function.

If **subprogram\_name** is overloaded, the pragma applies only to the most recent subprogram declaration.

**DEFAULT**
Specifies that the pragma applies to all subprograms in the package specification or object type specification (including the system-defined constructor for object types).

You can still declare the pragma for individual subprograms, overriding the DEFAULT pragma.

**RNDS**
Asserts that the subprogram reads no database state (does not query database tables).
WNDS

Asserts that the subprogram writes no database state (does not modify tables).

RNPS

Asserts that the subprogram reads no package state (does not reference the values of packaged variables)

You cannot specify RNPS if the subprogram invokes the SQLCODE or SQLERRM function.

WNPS

Asserts that the subprogram writes no package state (does not change the values of packaged variables).

You cannot specify WNPS if the subprogram invokes the SQLCODE or SQLERRM function.

TRUST

Asserts that the subprogram can be trusted not to violate one or more rules.

When you specify TRUST, the subprogram body is not checked for violations of the constraints listed in the pragma. The subprogram is trusted not to violate them. Skipping these checks can improve performance. TRUST is needed for functions written in C or Java that are invoked from PL/SQL, since PL/SQL cannot verify them at run time.

Usage Notes

A RESTRICT_REFERENCES pragma can appear only in a package specification or object type specification. Typically, this pragma is specified for functions. If a function calls procedures, specify the pragma for those procedures also.

To invoke a subprogram from parallel queries, you must specify all four constraints—RNDS, WNDS, RNPS, and WNPS. No constraint implies another.

Examples

- Example 6–48, "Invoking an Autonomous Function" on page 6-46
- Example 8–20, "RESTRICT_REFERENCES Pragma" on page 8-25

Related Topics

- AUTONOMOUS_TRANSACTION Pragma on page 13-6
- EXCEPTION_INIT Pragma on page 13-43
- SERIALLY_REUSABLE Pragma on page 13-120
- SQLCODE Function on page 13-125
- SQLERRM Function on page 13-126
- Controlling Side Effects of PL/SQL Subprograms on page 8-25
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.

Syntax

```
return_statement ::= RETURN expression
```

Keyword and Parameter Descriptions

`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 specification, which specifies the data type 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 1–19, "Creating a Package and Package Body" on page 1-21
- Example 2–23, "Using a Subprogram Name for Name Resolution" on page 2-21
- 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 13-75
- **RETURN Statement** on page 8-4
RETURNING INTO Clause

The RETURNING INTO clause specifies the variables in which to store the values returned by the statement to which the clause belongs. The variables can be either individual variables or collections. If the statement does not affect any rows, the values of the variables are undefined.

The static RETURNING INTO clause belongs to a DELETE, INSERT, or UPDATE statement. The dynamic RETURNING INTO clause belongs to an EXECUTE IMMEDIATE statement.

You cannot use the RETURNING INTO clause for remote or parallel deletes.

Syntax

\[
\text{static\_returning\_clause ::=}
\]

\[
\begin{align*}
\text{RETURNING} & \quad \rightarrow \quad \text{RETURN} & \quad \rightarrow \\
& \quad \rightarrow \quad \single\_row\_expression & \quad \rightarrow \quad \into\_clause \\
& \quad \rightarrow \quad \multiple\_row\_expression & \quad \rightarrow \quad \bulk\_collect\_into\_clause \\
\end{align*}
\]

\[
\text{dynamic\_returning\_clause ::=}
\]

\[
\begin{align*}
\text{RETURNING} & \quad \rightarrow \quad \into\_clause \\
& \quad \rightarrow \quad \bulk\_collect\_into\_clause \\
\end{align*}
\]

\[
\text{into\_clause ::=}
\]

\[
\begin{align*}
\text{INTO} & \quad \rightarrow \quad \text{variable\_name} \\
& \quad \rightarrow \quad \text{record\_name} \\
\end{align*}
\]

\[
\text{bulk\_collect\_into\_clause ::=}
\]

\[
\begin{align*}
\text{BULK} \quad \rightarrow \quad \text{COLLECT} \quad \rightarrow \quad \text{INTO} & \quad \rightarrow \\
& \quad \rightarrow \quad \text{collection\_name} \\
& \quad \rightarrow \quad \text{host\_array\_name} \\
\end{align*}
\]

Keyword and Parameter Descriptions

**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 Table, "Reducing Loop Overhead for DML Statements and Queries with Bulk SQL" on page 12-9.

**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 data type 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 a of bind argument cannot be a Boolean literal (TRUE, FALSE, or NULL). To pass the value NULL to the dynamic SQL statement, see Uninitialized Variable for NULL in **USING Clause** on page 7-4.

**Examples**
- Example 5–52, "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 12-21
Example 12–16, "Using FORALL with BULK COLLECT" on page 12-21

Related Topics
- EXECUTE IMMEDIATE Statement on page 13-47
- SELECT INTO Statement on page 13-116
- Using the EXECUTE IMMEDIATE Statement on page 7-2
%ROWTYPE Attribute

The %ROWTYPE attribute lets you declare a record that represents a row in a table or view. For each column in the referenced table or view, the record has a field with the same name and data type. To reference a field in the record, use \textit{record\_name.field\_name}. The record fields do not inherit the constraints or default values of the corresponding columns.

If the referenced item table or view changes, your declaration is automatically updated. You need not change your code when, for example, columns are added or dropped from the table or view.

Syntax

\begin{verbatim}
%rowtype_attribute ::=  
cursor_name
| cursor_variable_name %ROWTYPE | table_name |
\end{verbatim}

Keyword and Parameter Descriptions

\textit{cursor\_name}

An explicit cursor previously declared within the current scope.

\textit{cursor\_variable\_name}

A PL/SQL strongly typed cursor variable, previously declared within the current scope.

\textit{table\_name}

A database table or view that must be accessible when the declaration is elaborated.

Examples

- Example 1–6, "Using %ROWTYPE with an Explicit Cursor" on page 1-11
- Example 2–14, "Using %ROWTYPE to Declare a Record that Represents a Table Row" on page 2-15
- Example 2–15, "Declaring a Record that Represents a Subset of Table Columns" on page 2-15
- Example 2–16, "Declaring a Record that Represents a Row from a Join" on page 2-16
- Example 2–17, "Assigning One Record to Another, Correctly and Incorrectly" on page 2-16
- Example 2–18, "Using SELECT INTO for Aggregate Assignment" on page 2-17
- Example 3–15, "Column Constraints Inherited by Subtypes" on page 3-27
- Example 5–7, "Specifying Collection Element Types with %TYPE and %ROWTYPE" on page 5-9
Example 5–20, "Assigning Values to VARRAYs with Complex Data Types" on page 5-16
Example 5–42, "Declaring and Initializing Record Types" on page 5-31
Example 6–24, "Cursor Variable Returning a %ROWTYPE Variable" on page 6-24
Example 6–25, "Using the %ROWTYPE Attribute to Provide the Data Type" on page 6-24

Related Topics
- Constant on page 13-32
- Cursor Variable Declaration on page 13-39
- Explicit Cursor on page 13-53
- FETCH Statement on page 13-68
- Using the %ROWTYPE Attribute on page 2-15
SELECT INTO Statement

The SELECT INTO statement retrieves values from one or more database tables (as the SQL SELECT statement does) and stores them in either variables or a record (which the SQL SELECT statement does not do).

By default, the SELECT INTO statement retrieves one or more columns from a single row. With the BULK COLLECT INTO clause, this statement retrieves an entire result set at once.

Syntax

select_into_statement ::= 

```sql
SELECT
  [DISTINCT | UNIQUE | ALL]
  select_item,
  [BULK COLLECT INTO]
  variable_name,
  record_name
FROM
  table_reference
[THE (subquery)]
alias
rest_of_statement ;
```
**select_item ::=**

- `function_name`
- `numeric_literal`
- `schema_name . table_name`
- `schema_name . view_name`
- `sequence_name . CURRVAL`
- `sequence_name . NEXTVAL`
- `text`
- `NULL`

**table_reference ::=**

- `schema_name . table_name`
- `schema_name . view_name`
- `PARTITION (partition)`
- `SUBPARTITION (subpartition)`
- `@ dblink`

**Keyword and Parameter Descriptions**

**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 12-9.

**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`, the database 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.

**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 the database 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.

**view_name**
The name of a database view.

---

**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
SELECT INTO Statement

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 1-8
- Example 1–5, "Assigning Values to Variables as Parameters of a Subprogram" on page 1-8
- Example 1–12, "Using WHILE-LOOP for Control" on page 1-16
- 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 7–13, "Using Validation Checks to Guard Against SQL Injection" on page 7-16

Related Topics

- Assignment Statement on page 13-3
- FETCH Statement on page 13-68
- %ROWTYPE Attribute on page 13-114
- Selecting At Most One Row (SELECT INTO Statement) on page 6-16

See Also: Oracle Database SQL Language Reference for information about the SQL SELECT statement.
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.

This pragma is appropriate for packages that declare large temporary work areas that are used only once in the same session.

Syntax

```plaintext
seriously_reusablePragma ::= 
```

Keyword and Parameter Descriptions

**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.

**SERIALLY_REUSABLE**
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, the database generates an error.

Usage Notes

A SERIALLY_REUSABLE pragma can appear in the specification of a bodiless package, or in both the specification and body of a package. The pragma cannot appear only in the body of a package.

Examples

**Example 13–5 creating a serially reusable package**

```plaintext
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 13-43
- **INLINE Pragma** on page 13-82
- **RESTRICT_REFERENCES Pragma** on page 13-107

**See Also:** Oracle Database Advanced Application Developer’s Guide for more information about serially reusable PL/SQL packages
SQL (Implicit) Cursor Attribute

A SQL (implicit) cursor is opened by the database to process each SQL statement that is not associated with an explicit cursor. Every SQL (implicit) cursor has six attributes, each of which returns useful information about the execution of a data manipulation statement.

Syntax

```
sql_cursor ::= %FOUND | %ISOPEN | %NOTFOUND | %ROWCOUNT | %BULK_ROWCOUNT(index) | %BULK_EXCEPTIONS(index).ERROR_INDEX | %BULK_EXCEPTIONS(index).ERROR_CODE
```

Keyword and Parameter Descriptions

**%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 Database 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 the database 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 implicit cursor.

Usage Notes

You can use cursor attributes in procedural statements but not in SQL statements. Before the database 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 12-14.

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 deduce 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 6-14
- 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 12-14

Related Topics

- Explicit Cursor on page 13-53
- Cursor Attribute on page 13-37
- FORALL Statement on page 13-72
- Attributes of SQL Cursors on page 6-8
- Querying Data with PL/SQL on page 6-16
In an exception handler, the SQLCODE function returns the numeric code of the exception being handled. (Outside an exception handler, SQLCODE returns 0.)

For an exception that the database raises, the numeric code is the number of the associated Oracle Database error. This number is negative except for the error "no data found", whose numeric code is +100.

For a user-defined exception, the numeric code is either +1 (the default) or the Oracle Database error number associated with the exception by the EXCEPTION_INIT pragma.

A SQL statement cannot invoke SQLCODE. To use the value of SQLCODE in a SQL statement, assign it to a local variable first.

If a function invokes SQLCODE, and you use the RESTRICT_REFERENCES pragma to assert its purity, you cannot specify the constraints WNPS and RNPS.

Syntax

\texttt{sqlcode\_function ::=}

\texttt{SQLCODE}

Examples

- Example 11–11, "Displaying SQLCODE and SQLERRM" on page 11-15

Related Topics

- Block on page 13-8
- EXCEPTION_INITPragma on page 13-43
- Exception Handler on page 13-45
- RESTRICT\_REFERENCES Pragma on page 13-107
- SQLERRM Function on page 13-126
- Associating a PL/SQL Exception with a Number (EXCEPTION_INIT Pragma) on page 11-7
- Retrieving the Error Code and Error Message on page 11-15

See Also: Oracle Database Error Messages for a list of Oracle Database error messages and information about them, including their numbers
SQLERRM Function

The SQLERRM function returns the error message associated with an error number.

You cannot use SQLERRM directly in a SQL statement. Assign the value of SQLERRM to a local variable first.

If a function invokes SQLERRM, and you use the RESTRICT_REFERENCES pragma to assert its purity, you cannot specify the constraints WNPS and RNPS.

---

**Note:** DBMS_UTILTY.FORMAT_ERROR_STACK is recommended over SQLERRM, except when using the FORALL statement with its SAVE EXCEPTIONS clause. For more information, see Retrieving the Error Code and Error Message on page 11-15.

---

**Syntax**

```
sqlerrm_function ::= 
```

```
SQLERRM(error_number) 
```

**Keyword and Parameter Descriptions**

**error_number**

An expression whose value is an Oracle Database error number. For a list of Oracle Database error numbers, see Oracle Database Error Messages.

The default error number is the one associated with the current value of SQLCODE. Like SQLCODE, SQLERRM without error_number is useful only in an exception handler. Outside an exception handler, or if the value of error_number is zero, SQLERRM returns ORA-0000.

If the value of error_number is +100, SQLERRM returns ORA-01403.

If the value of error_number is a positive number other than +100, SQLERRM returns this message:

```
-error_number: non-ORACLE exception
```

If the value of error_number is a negative number whose absolute value is an Oracle Database error number, SQLERRM returns the error message associated with that error number. For example:

```
SQL> BEGIN 
  2     DBMS_OUTPUT.PUT_LINE('SQLERRM(-6511): ' || TO_CHAR(SQLERRM(-6511)));
  3     END;
  4     / 
```

```
SQLERRM(-6511): ORA-06511: PL/SQL: cursor already open
```

PL/SQL procedure successfully completed.

SQL>
If the value of `error_number` is a negative number whose absolute value is not an Oracle Database error number, `SQLERRM` returns this message:

```
ORA-error_number: Message error_number not found; product=RDBMS;
facility=ORA
```

For example:

```
SQL> BEGIN
  2  DBMS_OUTPUT.PUT_LINE('SQLERRM(-50000): ' || TO_CHAR(SQLERRM(-50000)));
  3  END;
  4 /
SQLERRM(-50000): ORA-50000: Message 50000 not found; product=RDBMS;
facility=ORA
```

PL/SQL procedure successfully completed.

Examples

- Example 11–11, "Displaying SQLCODE and SQLERRM" on page 11-15
- Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-16

Related Topics

- Block on page 13-8
- `EXCEPTION_INIT` Pragma on page 13-43
- `RESTRICT_REFERENCES` Pragma on page 13-107
- `SQLCODE` Function on page 13-125
- Retrieving the Error Code and Error Message on page 11-15

See Also: Oracle Database Error Messages for a list of Oracle Database error messages and information about them.
%TYPE Attribute

The %TYPE attribute lets you declare a constant, variable, field, or parameter to be of the same data type a previously declared variable, field, record, nested table, or database column. If the referenced item changes, your declaration is automatically updated.

An item declared with %TYPE (the referencing item) always inherits the data type of the referenced item. The referencing item inherits the constraints only if the referenced item is not a database column. The referencing item inherits the default value only if the referencing item is not a database column and does not have the NOT NULL constraint.

Syntax

%type_attribute ::=  

Keyword and Parameter Descriptions

**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.
%TYPE Attribute

**variable_name**
A variable, previously declared in the same scope.

**Examples**

- Example 1–7, "Using a PL/SQL Collection Type" on page 1-11
- Example 2–10, "Using %TYPE to Declare Variables of the Types of Other Variables" on page 2-13
- Example 2–11, "Using %TYPE Incorrectly with NOT NULL Referenced Type" on page 2-13
- Example 2–12, "Using %TYPE Correctly with NOT NULL Referenced Type" on page 2-13
- Example 2–13, "Using %TYPE to Declare Variables of the Types of Table Columns" on page 2-14
- Example 2–23, "Using a Subprogram Name for Name Resolution" on page 2-21
- Example 2–17, "Assigning One Record to Another, Correctly and Incorrectly" on page 2-16
- Example 3–15, "Column Constraints Inherited by Subtypes" on page 3-27
- 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 on page 13-32
- %ROWTYPE Attribute on page 13-114
- Variable on page 13-130
- Using the %TYPE Attribute on page 2-12
- Constraints and Default Values with Subtypes on page 3-26
Variable

A variable holds a value that can change.
A variable declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also assign an initial value and impose the NOT NULL constraint.

Syntax

variable_declaration ::= 

\[
\text{variable_name} \quad \text{datatype} \\
\text{expression} \quad \text{NOT} \quad \text{DEFAULT} \\
\]

(expression ::= on page 13-57)

datatype ::= 

Keyword and Parameter Descriptions

\textbf{collection_name}
A collection (associative array, nested table, or varray) previously declared within the current scope.

\textbf{collection_type_name}
A user-defined collection type defined using the data type specifier TABLE or VARRAY.
**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**
The value to be assigned to the variable when the declaration is elaborated. The value of expression must be of a data type that is compatible with the data type of the variable.

**NOT NULL**
A constraint that prevents the program from assigning a null value to the variable. Assigning a null to a variable defined as NOT NULL raises the predefined exception VALUE_ERROR.

**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 data type specifier RECORD.

**ref_cursor_type_name**
A user-defined cursor variable type, defined using the data type 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 data types as columns in the row.

**scalar_datatype_name**
A predefined scalar data type such as BOOLEAN, NUMBER, or VARCHAR2. Includes any qualifiers for size, precision, and character or byte semantics.
%TYPE
Represents the data type of a previously declared collection, cursor variable, field, object, record, database column, or variable.

variable_name
The name of the variable. For naming conventions, see Identifiers on page 2-4.

Usage Notes
Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL. Whether public or private, variables declared in a package specification are initialized only once for each session.

An initialization clause is required when declaring NOT NULL variables. If you use %ROWTYPE to declare a variable, initialization is not allowed.

Examples
- Example 1–2, "PL/SQL Variable Declarations" on page 1-7
- Example 1–3, "Assigning Values to Variables with the Assignment Operator" on page 1-7
- Example 1–4, "Using SELECT INTO to Assign Values to Variables" on page 1-8
- Example 2–15, "Declaring a Record that Represents a Subset of Table Columns" on page 2-15

Related Topics
- Assignment Statement on page 13-3
- Collection on page 13-22
- Expression on page 13-57
- %ROWTYPE Attribute on page 13-114
- %TYPE Attribute on page 13-128
- Declaring PL/SQL Variables on page 1-7
- Declarations on page 2-10
- Predefined PL/SQL Scalar Data Types and Subtypes on page 3-1
This chapter explains how to use the SQL statements that create, change, and drop stored PL/SQL units.

For instructions for reading the syntax diagrams in this chapter, see *Oracle Database SQL Language Reference*.

**CREATE [OR REPLACE] Statements**

Each of the following SQL statements creates a PL/SQL unit and stores it in the database:

- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement
- CREATE PROCEDURE Statement
- CREATE TRIGGER Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

Each of these CREATE statements has an optional OR REPLACE clause. Specify OR REPLACE to re-create an existing PL/SQL unit—that is, to change its declaration or definition without dropping it, re-creating it, and regranting object privileges previously granted on it. If you redefine a PL/SQL unit, the database recompiles it. None of these CREATE statements can appear in a PL/SQL block.

**ALTER Statements**

If you want to recompile an existing PL/SQL unit without re-creating it (without changing its declaration or definition), use one of the following SQL statements:

- ALTER FUNCTION Statement
- ALTER PACKAGE Statement
- ALTER PROCEDURE Statement
- ALTER TRIGGER Statement
- ALTER TYPE Statement

Two reasons to use an ALTER statement are:
To explicitly recompile a stored unit that has become invalid, thus eliminating the need for implicit run-time recompilation and preventing associated run-time compilation errors and performance overhead.

To recompile a unit with different compilation parameters.

For information about compilation parameters, see PL/SQL Units and Compilation Parameters on page 1-25.

The ALTER TYPE statement has additional uses. For details, see ALTER TYPE Statement on page 14-15.

DROP Statements
To drop an existing PL/SQL unit from the database, use one of the following SQL statements:

- DROP FUNCTION Statement
- DROP PACKAGE Statement
- DROP PROCEDURE Statement
- DROP TRIGGER Statement
- DROP TYPE Statement
- DROP TYPE BODY Statement
ALTER FUNCTION Statement

The ALTER FUNCTION statement explicitly recompiles an invalid standalone stored function. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Note: This statement does not change the declaration or definition of an existing function. To redeclare or redefine a standalone stored function, use the CREATE FUNCTION Statement on page 14-29 with the OR REPLACE clause.

Prerequisites

The function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

```sql
alter_function::=

ALTER FUNCTION schema . function

COMPILE DEBUG compiler_parameters_clause REUSE SETTINGS

compiler_parameters_clause::=

parameter_name = parameter_value
```

Keyword and Parameter Descriptions

**schema**
Specify the schema containing the function. If you omit `schema`, then the database assumes the function is in your own schema.

**function**
Specify the name of the function to be recompiled.

**COMPILE**
Specify COMPILE to cause the database to recompile the function. The COMPILE keyword is required. If the database does not compile the function successfully, then you can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the `REUSE SETTINGS` clause.
**DEBUG**

Specify `DEBUG` to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Specifying this clause has the same effect as specifying `PLSQL_DEBUG = TRUE` in the `compiler_parameters_clause`.

**compiler_parameters_clause**

Use this clause to specify a value for one of the PL/SQL persistent compiler parameters. The value of these initialization parameters at the time of compilation is stored with the unit's metadata. You can learn the value of such a parameter by querying the appropriate `*_PLSQL_OBJECT_SETTINGS` view. The PL/SQL persistent parameters are `PLSQL_OPTIMIZE_LEVEL`, `PLSQL_CODE_TYPE`, `PLSQL_DEBUG`, `PLSQL_WARNINGS`, `PLSQL_CCFLAGS`, and `NLS_LENGTH_SEMANTICS`.

You can specify each parameter only once in each statement. Each setting is valid only for the current library unit being compiled and does not affect other compilations in this session or system. To affect the entire session or system, you must set a value for the parameter using the `ALTER SESSION` or `ALTER SYSTEM` statement.

If you omit any parameter from this clause and you specify `REUSE_SETTINGS`, then if a value was specified for the parameter in an earlier compilation of this library unit, the database uses that earlier value. If you omit any parameter and either you do not specify `REUSE_SETTINGS` or no value has been specified for the parameter in an earlier compilation, then the database obtains the value for that parameter from the session environment.

**Restriction on the compiler_parameters_clause**

You cannot set a value for the `PLSQL_DEBUG` parameter if you also specify `DEBUG`, because both clauses set the `PLSQL_DEBUG` parameter, and you can specify a value for each parameter only once.

**See Also:**

- *Oracle Database Reference* for the valid values and semantics of each of these parameters
- *Conditional Compilation* on page 2-48 for more information about compilation parameters

**REUSE SETTINGS**

Specify `REUSE_SETTINGS` to prevent Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in this statement.

For backward compatibility, the database sets the persistently stored value of the `PLSQL_COMPILER_FLAGS` initialization parameter to reflect the values of the `PLSQL_CODE_TYPE` and `PLSQL_DEBUG` parameters that result from this statement.

**See Also:**

- *Oracle Database Reference* for the valid values and semantics of each of these parameters
- *Conditional Compilation* on page 2-48 for more information about compilation parameters
Example

Recompiling a Function: Example  To explicitly recompile the function get_bal owned by the sample user oe, issue the following statement:

```
ALTER FUNCTION oe.get_bal
    COMPILE;
```

If the database encounters no compilation errors while recompiling get_bal, then get_bal becomes valid. The database can subsequently execute it without recompiling it at run time. If recompiling get_bal results in compilation errors, then the database returns an error, and get_bal remains invalid.

The database also invalidates all objects that depend upon get_bal. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

Related Topics

- CREATE FUNCTION Statement on page 14-29
- DROP FUNCTION Statement on page 14-89
ALTER PACKAGE Statement

The ALTER PACKAGE statement explicitly recompiles a package specification, body, or both. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects together. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

**Note:** This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the CREATE PACKAGE Statement on page 14-39, or the CREATE PACKAGE BODY Statement on page 14-42 with the OR REPLACE clause.

Prerequisites

For you to modify a package, the package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

```
alter_package ::=  
  ALTER PACKAGE schema package compilation_clause [REUSE SETTINGS];
```

```
compilation_clause ::=  
  compiler_parameters_clause
```

```
compiler_parameters_clause ::=  
  (parameter_name = parameter_value)
```

Keyword and Parameter Descriptions

**schema**

Specify the schema containing the package. If you omit `schema`, then the database assumes the package is in your own schema.
**package**
Specify the name of the package to be recompiled.

**COMPILE**
You must specify **COMPILE** to recompile the package specification or body. The **COMPILE** keyword is required.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the **REUSE SETTINGS** clause.

If recompiling the package results in compilation errors, then the database returns an error and the body remains invalid. You can see the associated compiler error messages with the SQL*Plus command **SHOW ERRORS**.

*See Also:*  Recompiling a Package: Examples on page 14-8

**SPECIFICATION**
Specify **SPECIFICATION** to recompile only the package specification, regardless of whether it is invalid. You might want to recompile a package specification to check for compilation errors after modifying the specification.

When you recompile a package specification, the database invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

**BODY**
Specify **BODY** to recompile only the package body regardless of whether it is invalid. You might want to recompile a package body after modifying it. Recompiling a package body does not invalidate objects that depend upon the package specification.

When you recompile a package body, the database first recompiles the objects on which the body depends, if any of those objects are invalid. If the database recompiles the body successfully, then the body becomes valid.

**PACKAGE**
Specify **PACKAGE** to recompile both the package specification and the package body if one exists, regardless of whether they are invalid. This is the default. The recompilation of the package specification and body lead to the invalidation and recompilation of dependent objects as described for **SPECIFICATION** and **BODY**.

**DEBUG**
Specify **DEBUG** to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Specifying this clause has the same effect as specifying **PLSQL_DEBUG = TRUE** in the **compiler_parameters_clause**.

*See Also:*  Oracle Database PL/SQL Packages and Types Reference for information about debugging packages

**compiler_parameters_clause**
This clause has the same behavior for a package as it does for a function. See the **ALTER FUNCTION compiler_parameters_clause** on page 14-4.
**REUSE SETTINGS**

This clause has the same behavior for a package as it does for a function. See the `ALTER FUNCTION` clause `REUSE SETTINGS` on page 14-4.

**Examples**

**Recompiling a Package: Examples**  This statement explicitly recompiles the specification and body of the `hr.emp_mgmt` package. See `Creating a Package: Example` on page 14-40 for the example that creates this package.

```
ALTER PACKAGE emp_mgmt
    COMPILE PACKAGE;
```

If the database encounters no compilation errors while recompiling the `emp_mgmt` specification and body, then `emp_mgmt` becomes valid. The user `hr` can subsequently call or reference all package objects declared in the specification of `emp_mgmt` without run-time recompilation. If recompiling `emp_mgmt` results in compilation errors, then the database returns an error and `emp_mgmt` remains invalid.

The database also invalidates all objects that depend upon `emp_mgmt`. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

To recompile the body of the `emp_mgmt` package in the schema `hr`, issue the following statement:

```
ALTER PACKAGE hr.emp_mgmt
    COMPILE BODY;
```

If the database encounters no compilation errors while recompiling the package body, then the body becomes valid. The user `hr` can subsequently call or reference all package objects declared in the specification of `emp_mgmt` without run-time recompilation. If recompiling the body results in compilation errors, then the database returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of `emp_mgmt`, the database does not invalidate dependent objects.

**Related Topics**

- CREATE PACKAGE Statement on page 14-39
- DROP PACKAGE Statement on page 14-91
ALTER PROCEDURE Statement

The ALTER PROCEDURE statement explicitly recompiles a standalone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the ALTER PACKAGE Statement on page 14-6).

Note: This statement does not change the declaration or definition of an existing procedure. To redefine or redefine a standalone stored procedure, use the CREATE PROCEDURE Statement on page 14-46 with the OR REPLACE clause.

The ALTER PROCEDURE statement is very similar to the ALTER FUNCTION statement. See ALTER FUNCTION Statement on page 14-3 for more information.

Prerequisites

The procedure must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

```
alter_procedure ::= 

ALTER PROCEDURE schema . procedure 

COMPILE DEBUG compiler_parameters_clause REUSE SETTINGS 

compiler_parameters_clause ::= 

parameter_name = parameter_value 
```

Keyword and Parameter Descriptions

Schema

Specify the schema containing the procedure. If you omit `schema`, then the database assumes the procedure is in your own schema.

Procedure

Specify the name of the procedure to be recompiled.

Compile

Specify `COMPILE` to recompile the procedure. The `COMPILE` keyword is required. the database recompiles the procedure regardless of whether it is valid or invalid.
ALTER PROCEDURE Statement

- the database first recompiles objects upon which the procedure depends, if any of those objects are invalid.
- the database also invalidates any local objects that depend upon the procedure, such as procedures that call the recompiled procedure or package bodies that define procedures that call the recompiled procedure.
- If the database recompiles the procedure successfully, then the procedure becomes valid. If recompiling the procedure results in compilation errors, then the database returns an error and the procedure remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

**See Also:** Recompiling a Procedure: Example on page 14-10

### DEBUG

Specify `DEBUG` to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Specifying this clause is the same as specifying `PLSQL_DEBUG = TRUE` in the `compiler_parameters_clause`.

**See Also:** Oracle Database Advanced Application Developer’s Guide for information about debugging procedures

### compiler_parameters_clause

This clause has the same behavior for a procedure as it does for a function. See the `ALTER FUNCTION` `compiler_parameters_clause` on page 14-4.

### REUSE SETTINGS

This clause has the same behavior for a procedure as it does for a function. See the `ALTER FUNCTION` clause `REUSE SETTINGS` on page 14-4.

**Example**

**Recompiling a Procedure: Example** To explicitly recompile the procedure `remove_emp` owned by the user `hr`, issue the following statement:

```
ALTER PROCEDURE hr.remove_emp
   COMPILE;
```

If the database encounters no compilation errors while recompiling `remove_emp`, then `remove_emp` becomes valid. The database can subsequently execute it without recompiling it at run time. If recompiling `remove_emp` results in compilation errors, then the database returns an error and `remove_emp` remains invalid.

the database also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that call `remove_emp`. If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

### Related Topics

- **CREATE PROCEDURE Statement** on page 14-46
- **DROP PROCEDURE Statement** on page 14-93
ALTER TRIGGER Statement

The ALTER TRIGGER statement enables, disables, or compiles a database trigger.

**Note:** This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the CREATE TRIGGER Statement on page 14-51 with the OR REPLACE clause.

Prerequisites

The trigger must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

In addition, to alter a trigger on DATABASE, you must have the ADMINISTER database events system privilege.

**See Also:** CREATE TRIGGER Statement on page 14-51 for more information about triggers based on DATABASE triggers

Syntax

\[alter\_trigger::=\]

\[\]

- `ALTER TRIGGER` 
- `schema` 
- `trigger`

- `ENABLE` 
- `DISABLE`

- `RENAME TO` 
- `new_name`

- `COMPILE`

- `DEBUG`

- `compiler\_parameters\_clause`

- `REUSE`

- `SETTINGS`

**compiler\_parameters\_clause::=**

- `parameter\_name = parameter\_value`

Keyword and Parameter Descriptions

**schema**

Specify the schema containing the trigger. If you omit `schema`, then the database assumes the trigger is in your own schema.

**trigger**

Specify the name of the trigger to be altered.
**ENABLE | DISABLE**

Specify `ENABLE` to enable the trigger. You can also use the `ENABLE ALL TRIGGERS` clause of `ALTER TABLE` to enable all triggers associated with a table. See the `ALTER TABLE` statement in *Oracle Database SQL Language Reference*.

Specify `DISABLE` to disable the trigger. You can also use the `DISABLE ALL TRIGGERS` clause of `ALTER TABLE` to disable all triggers associated with a table.

**See Also:**
- Enabling Triggers: Example on page 14-14
- Disabling Triggers: Example on page 14-14

**RENAME Clause**

Specify `RENAME TO new_name` to rename the trigger. The database renames the trigger and leaves it in the same state it was in before being renamed.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the `USER_SOURCE`, `ALL_SOURCE`, and `DBA_SOURCE` data dictionary views. As a result, comments and formatting may change in the `TEXT` column of those views even though the trigger source did not change.

**COMPILE Clause**

Specify `COMPILE` to explicitly compile the trigger, whether it is valid or invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The database first recompiles objects upon which the trigger depends, if any of these objects are invalid. If the database compiles the trigger successfully, then the trigger becomes valid.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

If recompiling the trigger results in compilation errors, then the database returns an error and the trigger remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

**DEBUG**

Specify `DEBUG` to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Specifying this clause has the same effect as specifying `PLSQL_DEBUG = TRUE` in the `compiler_parameters_clause`.

**compiler_parameters_clause**

This clause has the same behavior for a trigger as it does for a function. See the `ALTER FUNCTION` `compiler_parameters_clause` on page 14-4.

**REUSE SETTINGS**

This clause has the same behavior for a trigger as it does for a function. See the `ALTER FUNCTION` clause `REUSE SETTINGS` on page 14-4.
Examples

Disabling Triggers: Example  The sample schema hr has a trigger named update_job_history created on the employees table. The trigger is fired whenever an UPDATE statement changes an employee’s job_id. The trigger inserts into the job_history table a row that contains the employee’s ID, begin and end date of the last job, and the job ID and department.

When this trigger is created, the database enables it automatically. You can subsequently disable the trigger with the following statement:

```
ALTER TRIGGER update_job_history DISABLE;
```

When the trigger is disabled, the database does not fire the trigger when an UPDATE statement changes an employee’s job.

Enabling Triggers: Example  After disabling the trigger, you can subsequently enable it with the following statement:

```
ALTER TRIGGER update_job_history ENABLE;
```

After you reenable the trigger, the database fires the trigger whenever an employee’s job changes as a result of an UPDATE statement. If an employee’s job is updated while the trigger is disabled, then the database does not automatically fire the trigger for this employee until another transaction changes the job_id again.

Related Topics

- CREATE TRIGGER Statement on page 14-51
- DROP TRIGGER Statement on page 14-94
ALTER TYPE Statement

The ALTER TYPE statement adds or drops member attributes or methods. You can change the existing properties (FINAL or INSTANTIABLE) of an object type, and you can modify the scalar attributes of the type.

You can also use this statement to recompile the specification or body of the type or to change the specification of an object type by adding new object member subprogram specifications.

Prerequisites

The object type must be in your own schema and you must have CREATE TYPE or CREATE ANY TYPE system privilege, or you must have ALTER ANY TYPE system privileges.

Syntax

\[
\text{ALTER TYPE} \quad \text{schema} . \text{type} \\
\text{compile_type_clause} \\
\text{replace_type_clause} \\
\text{alter_method_spec} \\
\text{alter_attribute_definition} \\
\text{alter_collection_clauses} \\
\text{dependent_handling_clause} \\
\text{NOT} \quad \text{INSTANTIABLE} \\
\text{FINAL} \\
\]

\[
\text{COMPILE} \quad \text{DEBUG} \quad \text{SPECIFICATION} \\
\text{body} \\
\text{compiler_parameters_clause} \\
\text{REUSE} \quad \text{SETTINGS} \\
\text{parameter_name} = \text{parameter_value} \\
\]

\[
\text{compile_type_clause} ::= \\
\text{replace_type_clause} ::= \\
\text{alter_method_spec} ::= \\
\text{alter_attribute_definition} ::= \\
\text{alter_collection_clauses} ::= \\
\text{dependent_handling_clause} ::= \\
\text{parameter_name} = \text{parameter_value} \\
\]
replace_type_clause ::= 

invoker_rights_clause ::= 

element_spec ::= 

inheritance_clauses ::= 

subprogram_spec ::= 

procedure_spec ::= 

function_spec ::=
**constructor_spec::=**

- FINAL
- INSTANTIABLE
- CONSTRUCTOR
- FUNCTION
- datatype

- SELF
- IN
- OUT
- datatype

- parameter
- datatype

- RETURN
- SELF
- AS
- RESULT

- IS
- AS
- call_spec

**map_order_function_spec::=**

- MAP
- ORDER
- MEMBER
- function_spec

*(function_spec::= on page 14-17)*

**pragma_clause::=**

- PRAGMA
- RESTRICT_REFERENCES
- method_name
- DEFAULT

- RNDs
- WNDS
- RNPS
- WNPS
- TRUST

**alter_method_spec::=**

- ADD
- DROP

- map_order_function_spec
- subprogram_spec

*(map_order_function_spec::= on page 14-18, subprogram_spec::= on page 14-17)*
**alter_attribute_definition::=**

```
ADD MODIFY ATTRIBUTE attribute datatype
DROP ATTRIBUTE attribute datatype
```

**alter_collection_clauses::=**

```
MODIFY LIMIT integer ELEMENT TYPE datatype
```

**dependent_handling_clause::=**

```
INVALIDATE NOT INCLUDING TABLE DATA CONVERT TO SUBSTITUTABLE FORCE exceptions_clause
CASCADE
```

**exceptions_clause::=**

```
EXCEPTIONS INTO schema table
```

### Keyword and Parameter Descriptions

**schema**
Specify the schema that contains the type. If you omit `schema`, then the database assumes the type is in your current schema.

**type**
Specify the name of an object type, a nested table type, or a varray type.

**compile_type_clause**
Specify `COMPIL`e to compile the object type specification and body. This is the default if neither `SPECIFICATION` nor `BODY` is specified.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

If recompiling the type results in compilation errors, then the database returns an error and the type remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`. 
See Also:

- Recompiling a Type: Example on page 14-28
- Recompiling a Type Specification: Examples

DEBUG
Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Specifying this clause has the same effect as specifying PLSQL_DEBUG = TRUE in the compiler_parameters_clause.

SPECIFICATION
Specify SPECIFICATION to compile only the object type specification.

BODY
Specify BODY to compile only the object type body.

compiler_parameters_clause
This clause has the same behavior for a type as it does for a function. See the ALTER FUNCTION compiler_parameters_clause on page 14-4.

REUSE SETTINGS
This clause has the same behavior for a type as it does for a function. See the ALTER FUNCTION clause REUSE SETTINGS on page 14-4.

replace_type_clause
The REPLACE clause lets you add new member subprogram specifications. This clause is valid only for object types, not for nested tables or varrays.

attribute
Specify an object attribute name. Attributes are data items with a name and a type specifier that form the structure of the object.

element_spec
Specify the elements of the redefined object.

inheritance_clauses
The inheritance_clauses have the same semantics in CREATE TYPE and ALTER TYPE statements.

subprogram_spec
The MEMBER and STATIC clauses let you specify for the object type a function or procedure subprogram which is referenced as an attribute.

You must specify a corresponding method body in the object type body for each procedure or function specification.

See Also:

- CREATE TYPE Statement on page 14-65 for a description of the difference between member and static methods, and for examples
- CREATE TYPE BODY Statement on page 14-83
- Overloading PL/SQL Subprogram Names on page 8-12 for information about overloading subprogram names within a package
**procedure_spec** Enter the specification of a procedure subprogram.

**function_spec** Enter the specification of a function subprogram.

**pragma_clause** The *pragma_clause* is a compiler directive that denies member functions read/write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

**Note:** This clause has been deprecated. Oracle recommends against using this clause unless you must do so for backward compatibility of your applications. The database now runs purity checks at run time. If you must use this clause for backward compatibility of your applications, see its description in `CREATE TYPE Statement` on page 14-65.

**Restriction on Pragmas** The *pragma_clause* is not valid when dropping a method.

**See Also:** Oracle Database Advanced Application Developer’s Guide for more information about pragmas

**map_order_function_spec** You can declare either one *MAP* method or one *ORDER* method, regardless how many *MEMBER* or *STATIC* methods you declare. However, a subtype can override a *MAP* method if the supertype defines a *NOT FINAL MAP* method. If you declare either method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method must be specified to determine the equality of two object types.

- **For MAP**, specify a member function (*MAP* method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. The database uses the ordering for comparison conditions and *ORDER BY* clauses.

  If *type* will be referenced in queries involving sorts (through *ORDER BY*, *GROUP BY*, *DISTINCT*, or *UNION* clauses) or joins, and you want those queries to be parallelized, then you must specify a *MAP* member function.

  If the argument to the *MAP* method is null, then the *MAP* method returns null and the method is not invoked.

  An object specification can contain only one *MAP* method, which must be a function. The result type must be a predefined SQL scalar type, and the *MAP* function can have no arguments other than the implicit *SELF* argument.

  A subtype cannot define a new *MAP* method. However, it can override an inherited *MAP* method.

- **For ORDER**, specify a member function (*ORDER* method) that takes an instance of an object as an explicit argument and the implicit *SELF* argument and returns either a negative, zero, or positive integer. The negative, zero, or positive value indicates that the implicit *SELF* argument is less than, equal to, or greater than the explicit argument.

  If either argument to the *ORDER* method is null, then the *ORDER* method returns null and the method is not invoked.
When instances of the same object type definition are compared in an ORDER BY clause, the ORDER method function is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

A subtype cannot define an ORDER method, nor can it override an inherited ORDER method.

**invoker_rights_clause**

Specifies the AUTHID property of the member functions and procedures of the object type. For information about the AUTHID property, see "Using Invoker's Rights or Definer's Rights (AUTHID Clause)" on page 8-18.

**Restriction on Invoker’s Rights** You can specify this clause only for an object type, not for a nested table or varray.

**AUTHID CURRENT_USER Clause** Specify CURRENT_USER if you want the member functions and procedures of the object type to execute with the privileges of CURRENT_USER. This clause creates an invoker’s rights type.

You must specify this clause to maintain invoker’s rights status for the type if you created it with this status. Otherwise the status will revert to definer’s rights.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the type resides.

**AUTHID DEFINER Clause** Specify DEFINER if you want the member functions and procedures of the object type to execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default.

**See Also:** Using Invoker’s Rights or Definer’s Rights (AUTHID Clause) on page 8-18

**alter_method_spec**

The alter_method_spec lets you add a method to or drop a method from type. the database disables any function-based indexes that depend on the type.

In one ALTER TYPE statement you can add or drop multiple methods, but you can reference each method only once.

**ADD** When you add a method, its name must not conflict with any existing attributes in its type hierarchy.

**See Also:** Adding a Member Function: Example on page 14-27

**DROP** When you drop a method, the database removes the method from the target type.

**Restriction on Dropping Methods** You cannot drop from a subtype a method inherited from its supertype. Instead you must drop the method from the supertype.

**subprogram_spec** The MEMBER and STATIC clauses let you add a procedure subprogram to or drop it from the object type.
Restriction on Subprograms  You cannot define a STATIC method on a subtype that redefines a MEMBER method in its supertype, or vice versa.

map_order_function_spec  If you declare either a MAP or ORDER method, then you can compare object instances in SQL.

Restriction on MAP and ORDER Methods  You cannot add an ORDER method to a subtype.

alter_attribute_definition
The alter_attribute_definition clause lets you add, drop, or modify an attribute of an object type. In one ALTER TYPE statement, you can add, drop, or modify multiple member attributes or methods, but you can reference each attribute or method only once.

ADD ATTRIBUTE  The name of the new attribute must not conflict with existing attributes or methods in the type hierarchy. The database adds the new attribute to the end of the locally defined attribute list.

If you add the attribute to a supertype, then it is inherited by all of its subtypes. In subtypes, inherited attributes always precede declared attributes. Therefore, you might need to update the mappings of the implicitly altered subtypes after adding an attribute to a supertype.

See Also:  Adding a Collection Attribute: Example on page 14-27

DROP ATTRIBUTE  When you drop an attribute from a type, the database drops the column corresponding to the dropped attribute as well as any indexes, statistics, and constraints referencing the dropped attribute.

You need not specify the data type of the attribute you are dropping.

Restrictions on Dropping Type Attributes  Dropping type attributes is subject to the following restrictions:

- You cannot drop an attribute inherited from a supertype. Instead you must drop the attribute from the supertype.
- You cannot drop an attribute that is part of a partitioning, subpartitioning, or cluster key.
- You cannot drop an attribute of a primary-key-based object identifier of an object table or a primary key of an index-organized table.
- You cannot drop all of the attributes of a root type. Instead you must drop the type. However, you can drop all of the locally declared attributes of a subtype.

MODIFY ATTRIBUTE  This clause lets you modify the data type of an existing scalar attribute. For example, you can increase the length of a VARCHAR2 or RAW attribute, or you can increase the precision or scale of a numeric attribute.

Restriction on Modifying Attributes  You cannot expand the size of an attribute referenced in a function-based index, domain index, or cluster key.

[NOT] FINAL
Use this clause to indicate whether any further subtypes can be created for this type:

- Specify FINAL if no further subtypes can be created for this type.
Specify **NOT FINAL** if further subtypes can be created under this type.

If you change the property between **FINAL** and **NOT FINAL**, then you must specify the `CASCADE` clause of the `dependent_handling_clause` on page 14-25 to convert data in dependent columns and tables.

- If you change a type from **NOT FINAL** to **FINAL**, then you must specify `CASCADE [INCLUDING TABLE DATA]`. You cannot defer data conversion with `CASCADE NOT INCLUDING TABLE DATA`.

- If you change a type from **FINAL** to **NOT FINAL**, then:
  - Specify `CASCADE INCLUDING TABLE DATA` if you want to create new substitutable tables and columns of that type, but you are not concerned about the substitutability of the existing dependent tables and columns. The database marks all existing dependent columns and tables **NOT SUBSTITUTABLE AT ALL LEVELS**, so you cannot insert the new subtype instances of the altered type into these existing columns and tables.

  - Specify `CASCADE CONVERT TO SUBSTITUTABLE` if you want to create new substitutable tables and columns of the type and also store new subtype instances of the altered type in existing dependent tables and columns. The database marks all existing dependent columns and tables **SUBSTITUTABLE AT ALL LEVELS** except those that are explicitly marked **NOT SUBSTITUTABLE AT ALL LEVELS**.

**See Also:** *Oracle Database Object-Relational Developer’s Guide* for a full discussion of object type evolution

**Restriction on **FINAL** You cannot change a user-defined type from **NOT FINAL** to **FINAL** if the type has any subtypes.

**[NOT] INSTANTIABLE**

Use this clause to indicate whether any object instances of this type can be constructed:

- Specify **INSTANTIABLE** if object instances of this type can be constructed.

- Specify **NOT INSTANTIABLE** if no constructor (default or user-defined) exists for this object type. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement).

**Restriction on **NOT INSTANTIABLE** You cannot change a user-defined type from **INSTANTIABLE** to **NOT INSTANTIABLE** if the type has any table dependents.

**alter_collection_clauses**

These clauses are valid only for collection types.

**MODIFY LIMIT** `integer` This clause lets you increase the number of elements in a varray. It is not valid for nested tables. Specify an integer greater than the current maximum number of elements in the varray.

**See Also:**  *Increasing the Number of Elements of a Collection Type: Example* on page 14-27

**ELEMENT TYPE** `datatype`  This clause lets you increase the precision, size, or length of a scalar data type of a varray or nested table. This clause is not valid for collections of object types.
For a collection of `NUMBER`, you can increase the precision or scale.

For a collection of `RAW`, you can increase the maximum size.

For a collection of `VARCHAR2` or `NVARCHAR2`, you can increase the maximum length.

See Also: Increasing the Length of a Collection Type: Example on page 14-27

dependent_handling_clause

The `dependent_handling_clause` lets you instruct the database how to handle objects that are dependent on the modified type. If you omit this clause, then the `ALTER TYPE` statement will terminate if `type` has any dependent type or table.

INVALIDATE Clause

Specify `INVALIDATE` to invalidate all dependent objects without any checking mechanism.

Note: the database does not validate the type change, so you should use this clause with caution. For example, if you drop an attribute that is a partitioning or cluster key, then you will be unable to write to the table.

CASCADE Clause

Specify the `CASCADE` clause if you want to propagate the type change to dependent types and tables. the database terminates the statement if any errors are found in the dependent types or tables unless you also specify `FORCE`.

If you change the property of the type between `FINAL` and `NOT FINAL`, then you must specify this clause to convert data in dependent columns and tables. See [NOT] FINAL on page 14-24.

INCLUDING TABLE DATA

Specify `INCLUDING TABLE DATA` to convert data stored in all user-defined columns to the most recent version of the column type. This is the default.

Note: You must specify this clause if your column data is in Oracle database version 8.0 image format. This clause is also required if you are changing the type property between `FINAL` and `NOT FINAL`.

For each attribute added to the column type, the database adds a new attribute to the data and initializes it to null.

For each attribute dropped from the referenced type, the database removes the corresponding attribute data from each row in the table.

If you specify `INCLUDING TABLE DATA`, then all of the tablespaces containing the table data must be in read/write mode.

If you specify `NOT INCLUDING TABLE DATA`, then the database upgrades the metadata of the column to reflect the changes to the type but does not scan the dependent column and update the data as part of this `ALTER TYPE` statement. However, the dependent column data remains accessible, and the results of subsequent queries of the data will reflect the type modifications.
ALTER TYPE Statement

See Also: Oracle Database Object-Relational Developer’s Guide for more information about the implications of not including table data when modifying type attribute

CONVERT TO SUBSTITUTABLE Specify this clause if you are changing the type from FINAL to NOT FINAL and you want to create new substitutable tables and columns of the type and also store new subtype instances of the altered type in existing dependent tables and columns. See [NOT] FINAL on page 14-24 for more information.

exceptions_clause Specify FORCE if you want the database to ignore the errors from dependent tables and indexes and log all errors in the specified exception table. The exception table must already have been created by executing the DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE procedure.

Examples

Adding a Member Function: Example The following example uses the data_typ1 object type. See Object Type Examples on page 14-78 for the example that creates this object type. A method is added to data_typ1 and its type body is modified to correspond. The date formats are consistent with the order_date column of the oe.orders sample table:

```
ALTER TYPE data_typ1
   ADD MEMBER FUNCTION qtr(der_qtr DATE)
   RETURN CHAR CASCADE;

CREATE OR REPLACE TYPE BODY data_typ1 IS
   MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
      BEGIN
         RETURN (year + invent);
      END;
   MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
      BEGIN
         IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
            RETURN 'FIRST';
         ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
            RETURN 'SECOND';
         ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
            RETURN 'THIRD';
         ELSE
            RETURN 'FOURTH';
         END IF;
      END IF;
   END;
/
```

Adding a Collection Attribute: Example The following example adds the author attribute to the textdoc_tab object column of the text table. See Object Type Examples on page 14-78 for the example that creates the underlying textdoc_typ type.

```
CREATE TABLE text (
   doc_id       NUMBER,
   description  textdoc_tab)
NESTED TABLE description STORE AS text_store;

ALTER TYPE textdoc_typ
   ADD ATTRIBUTE (author VARCHAR2) CASCADE;
```

14-24 Oracle Database PL/SQL Language Reference
The CASCADE keyword is required because both the textdoc_tab and text table are dependent on the textdoc_typ type.

**Increasing the Number of Elements of a Collection Type: Example** The following example increases the maximum number of elements in the varray phone_list_typ_demo. See Object Type Examples on page 14-78 for the example that creates this type.

```sql
ALTER TYPE phone_list_typ_demo
    MODIFY LIMIT 10 CASCADE;
```

**Increasing the Length of a Collection Type: Example** The following example increases the length of the varray element type phone_list_typ:

```sql
ALTER TYPE phone_list_typ
    MODIFY ELEMENT TYPE VARCHAR(64) CASCADE;
```

**Recompiling a Type: Example** The following example recompiles type cust_address_typ in the hr schema:

```sql
ALTER TYPE cust_address_typ2 COMPILE;
```

**Recompiling a Type Specification: Example** The following example compiles the type specification of link2.

```sql
CREATE TYPE link1 AS OBJECT
    (a NUMBER);
/
CREATE TYPE link2 AS OBJECT
    (a NUMBER,
     b link1,
     MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);
/
CREATE TYPE BODY link2 AS
    MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS
    BEGIN
        dbms_output.put_line(c1);
        RETURN c1;
    END;
END;
/
```

In the following example, both the specification and body of link2 are invalidated because link1, which is an attribute of link2, is altered.

```sql
ALTER TYPE link1 ADD ATTRIBUTE (b NUMBER) INVALIDATE;
```

You must recompile the type by recompiling the specification and body in separate statements:

```sql
ALTER TYPE link2 COMPILE SPECIFICATION;
```

```sql
ALTER TYPE link2 COMPILE BODY;
```

Alternatively, you can compile both specification and body at the same time:

```sql
ALTER TYPE link2 COMPILE;
```

### Related Topics
- CREATE TYPE Statement on page 14-65
ALTER TYPE Statement

- CREATE TYPE BODY Statement on page 14-83
- DROP TYPE Statement on page 14-95
CREATE FUNCTION Statement

The CREATE FUNCTION statement creates or replaces a standalone stored function or a call specification.

A standalone stored function is a function (a subprogram that returns a single value) that is stored in the database.

Note: A standalone stored function that you create with the CREATE FUNCTION statement is different from a function that you declare and define in a PL/SQL block or package. For information about the latter, see Function Declaration and Definition on page 13-75.

A call specification declares a Java method or a third-generation language (3GL) routine so that it can be called from PL/SQL. You can also use the SQL CALL statement to call such a method or routine. The call specification tells the database which Java method, or which named function in which shared library, to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

Note: To be callable from SQL statements, a stored function must obey certain rules that control side effects. See Controlling Side Effects of PL/SQL Subprograms on page 8-25.

Prerequisites

To create or replace a standalone stored function in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a standalone stored function in another user's schema, you must have the CREATE ANY PROCEDURE system privilege.

To invoke a call specification, you may need additional privileges, for example, EXECUTE privileges on a C library for a C call specification.

To embed a CREATE FUNCTION statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

See Also: For more information about such prerequisites:

- Oracle Database Advanced Application Developer’s Guide
- Oracle Database Java Developer’s Guide
Syntax

create_function ::= 

create_function ::= CREATE OR REPLACE FUNCTION schema.function_name (parameter_declaration) RETURN datatype invoker_rights_clause ...

invoker_rights_clause ::= 

invoker_rights_clause ::= AUTHID CURRENT_USER | AUTHID DEFINER

parallel_enable_clause ::= 

parallel_enable_clause ::= PARALLEL_ENABLE 

streaming_clause ::= 

streaming_clause ::= ORDER BY expr | CLUSTER BY expr
call_spec ::= 

Java_declaration ::= 

C_declaration ::= 

Keyword and Parameter Descriptions

**OR REPLACE**
Specify **OR REPLACE** to re-create the function if it already exists. Use this clause to change the definition of an existing function without dropping, re-creating, and regranting object privileges previously granted on the function. If you redefine a function, then the database recompiles it.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, then the database marks the indexes **DISABLED**.

**schema**
Specify the schema to contain the function. If you omit *schema*, then the database creates the function in your current schema.

**function_name**
Specify the name of the function to be created.

**RETURN datatype**
For *datatype*, specify the data type of the return value of the function. The return value can have any data type supported by PL/SQL.

**Note:** Oracle SQL does not support calling of functions with Boolean parameters or returns. Therefore, if your user-defined functions will be called from SQL statements, you must design them to return numbers (0 or 1) or character strings ("TRUE" or "FALSE").
The data type cannot specify a length, precision, or scale. The database derives the length, precision, or scale of the return value from the environment from which the function is called.

If the return type is ANYDATASET and you intend to use the function in the FROM clause of a query, then you must also specify the PIPELINED clause and define a describe method (ODCITableDescribe) as part of the implementation type of the function.

You cannot constrain this data type (with NOT NULL, for example).

**See Also:**
- Chapter 3, "PL/SQL Data Types," for information about PL/SQL data types
- Oracle Database Data Cartridge Developer’s Guide for information about defining the ODCITableDescribe function

**invoker_rights_clause**

Specifies the AUTHID property of the member functions and procedures of the object type. For information about the AUTHID property, see "Using Invoker's Rights or Definer's Rights (AUTHID Clause)" on page 8-18.

**AUTHID Clause**

- Specify CURRENT_USER if you want the function to execute with the privileges of CURRENT_USER. This clause creates an invoker’s rights function.

  This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the function resides.

- Specify DEFINER if you want the function to execute with the privileges of the owner of the schema in which the function resides, and that external names resolve in the schema where the function resides. This is the default and creates a definer’s rights function.

  **See Also:**
  - Using Invoker's Rights or Definer's Rights (AUTHID Clause) on page 8-18 for more information about the AUTHID clause
  - Oracle Database Security Guide for information about invoker’s rights and definer’s rights types

**DETERMINISTIC**

Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its parameters.

You must specify this keyword if you intend to call the function in the expression of a function-based index or from the query of a materialized view that is marked REFRESH FAST or ENABLE QUERY REWRITE. When the database encounters a deterministic function in one of these contexts, it attempts to use previously calculated results when possible rather than reexecuting the function. If you subsequently change the semantics of the function, then you must manually rebuild all dependent function-based indexes and materialized views.

Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the return result of the function. The
results of doing so will not be captured if the database chooses not to reexecute the function.

The following semantic rules govern the use of the `DETERMINISTIC` clause:

- You can declare a top-level subprogram `DETERMINISTIC`.
- You can declare a package-level subprogram `DETERMINISTIC` in the package specification but not in the package body.
- You cannot declare `DETERMINISTIC` a private subprogram (declared inside another subprogram or inside a package body).
- A `DETERMINISTIC` subprogram can call another subprogram whether the called program is declared `DETERMINISTIC` or not.

**See Also:**
- Oracle Database Data Warehousing Guide for information about materialized views
- Oracle Database SQL Language Reference for information about function-based indexes

**parallel_enable_clause**

`PARALLEL_ENABLE` is an optimization hint indicating that the function can be executed from a parallel execution server of a parallel query operation. The function should not use session state, such as package variables, as those variables are not necessarily shared among the parallel execution servers.

- The optional `PARTITION` argument BY clause is used only with functions that have a `REF CURSOR` argument type. It lets you define the partitioning of the inputs to the function from the `REF CURSOR` argument.

Partitioning the inputs to the function affects the way the query is parallelized when the function is used as a table function in the `FROM` clause of the query. `ANY` indicates that the data can be partitioned randomly among the parallel execution servers. Alternatively, you can specify `RANGE` or `HASH` partitioning on a specified column list.

- The optional `streaming_clause` lets you order or cluster the parallel processing by a specified column list.
  - `ORDER BY` indicates that the rows on a parallel execution server must be locally ordered.
  - `CLUSTER BY` indicates that the rows on a parallel execution server must have the same key values as specified by the `column_list`.
  - `expr` identifies the `REF CURSOR` parameter name of the table function on which partitioning was specified, and on whose columns you are specifying ordering or clustering for each slave in a parallel query execution.

The columns specified in all of these optional clauses refer to columns that are returned by the `REF CURSOR` argument of the function.

**See Also:** For more information about user-defined aggregate functions:
- Oracle Database Advanced Application Developer’s Guide
- Oracle Database Data Cartridge Developer’s Guide
PIPEDINED { IS | USING }

Specify PIPEDINED to instruct the database to return the results of a table function iteratively. A table function returns a collection type (a nested table or varray). You query table functions by using the TABLE keyword before the function name in the FROM clause of the query. For example:

```
SELECT * FROM TABLE(function_name(...))
```

the database then returns rows as they are produced by the function.

- If you specify the keyword PIPEDINED alone (PIPEDINED IS ...), then the PL/SQL function body should use the PIPE keyword. This keyword instructs the database to return single elements of the collection out of the function, instead of returning the whole collection as a single value.

- You can specify the PIPEDINED USING implementation_type clause if you want to predefine an interface containing the start, fetch, and close operations. The implementation type must implement the ODCITable interface and must exist at the time the table function is created. This clause is useful for table functions that will be implemented in external languages such as C++ and Java.

If the return type of the function is ANYDATASET, then you must also define a describe method (ODCITableDescribe) as part of the implementation type of the function.

See Also:

- Performing Multiple Transformations with Pipelined Table Functions on page 12-34
- Oracle Database Data Cartridge Developer’s Guide for information about ODCI routines

AGGREGATE USING

Specify AGGREGATE USING to identify this function as an aggregate function, or one that evaluates a group of rows and returns a single row. You can specify aggregate functions in the select list, HAVING clause, and ORDER BY clause.

When you specify a user-defined aggregate function in a query, you can treat it as an analytic function (one that operates on a query result set). To do so, use the OVER analytic_clause syntax available for built-in analytic functions. See Oracle Database SQL Language Reference for syntax and semantics of analytic functions.

In the USING clause, specify the name of the implementation type of the function. The implementation type must be an object type containing the implementation of the ODCIAggregate routines. If you do not specify schema, then the database assumes that the implementation type is in your own schema.

Restriction on Creating Aggregate Functions  If you specify this clause, then you can specify only one input argument for the function.

See Also:  Oracle Database Data Cartridge Developer’s Guide for information about ODCI routines

body

The required executable part of the function and, optionally, the exception-handling part of the function.
**DECLARE SECTION**

The optional declarative part of the function. Declarations are local to the function, can be referenced in `body`, and cease to exist when the function completes execution.

**CALLSPEC**

Use the `CALLSPEC` to map a Java or C method name, parameter types, and return type to their SQL counterparts. In `JAVA_DECLARATION`, `'string'` identifies the Java implementation of the method.

**SEE ALSO**

- Oracle Database Java Developer’s Guide
- Oracle Database Advanced Application Developer’s Guide for information about calling external procedures

**EXTERNAL**

In earlier releases, `EXTERNAL` was an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle recommends that you use the `LANGUAGE C` syntax.

**EXAMPLES**

### Creating a Function: Examples

The following statement creates the function `get_bal` on the sample table `oe.orders`:

```sql
CREATE FUNCTION get_bal(acc_no IN NUMBER) RETURN NUMBER IS acc_bal NUMBER(11,2);
BEGIN
    SELECT order_total INTO acc_bal
    FROM orders
    WHERE customer_id = acc_no;
    RETURN(acc_bal);
END;
/
```

The `get_bal` function returns the balance of a specified account.

When you call the function, you must specify the argument `acc_no`, the number of the account whose balance is sought. The data type of `acc_no` is `NUMBER`.

The function returns the account balance. The `RETURN` clause of the `CREATE FUNCTION` statement specifies the data type of the return value to be `NUMBER`.

The function uses a `SELECT` statement to select the `balance` column from the row identified by the argument `acc_no` in the `orders` table. The function uses a `RETURN` statement to return this value to the environment in which the function is called.

The function created in the preceding example can be used in a SQL statement. For example:

```sql
SELECT get_bal(165) FROM DUAL;
```

```
GET_BAL(165)
------------
2519
```
The hypothetical following statement creates a PL/SQL standalone function `get_val` that registers the C routine `c_get_val` as an external function. (The parameters have been omitted from this example.)

```sql
CREATE FUNCTION get_val
    ( x_val IN NUMBER,
      y_val IN NUMBER,
      image IN LONG RAW )
RETURN BINARY_INTEGER AS LANGUAGE C
    NAME "c_get_val"
    LIBRARY c_utils
    PARAMETERS (...) ;
```

Creating Aggregate Functions: Example  The next statement creates an aggregate function called `SecondMax` to aggregate over number values. It assumes that the object type `SecondMaxImpl` routines contains the implementations of the ODCIAggregate routines:

```sql
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER
    PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

**See Also:** *Oracle Database Data Cartridge Developer's Guide* for the complete implementation of type and type body for `SecondMaxImpl`

You would use such an aggregate function in a query like the following statement, which queries the sample table `hr.employees`:

```sql
SELECT SecondMax(salary) 'SecondMax', department_id
FROM employees
GROUP BY department_id
HAVING SecondMax(salary) > 9000
ORDER BY 'SecondMax', department_id;
```

<table>
<thead>
<tr>
<th>SecondMax</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>13500</td>
<td>80</td>
</tr>
<tr>
<td>17000</td>
<td>90</td>
</tr>
</tbody>
</table>

Using a Packaged Procedure in a Function: Example  The following statement creates a function that uses a `DBMS_LOB.GETLENGTH` procedure to return the length of a CLOB column:

```sql
CREATE OR REPLACE FUNCTION text_length(a CLOB)
    RETURN NUMBER DETERMINISTIC IS
    BEGIN
      RETURN DBMS_LOB.GETLENGTH(a);
    END;
```

**See Also:** *Oracle Database SQL Language Reference* for an example of using this function to create a function-based index

Related Topics

- ALTER FUNCTION Statement on page 14-3
- CREATE PROCEDURE Statement on page 14-46
- DROP FUNCTION Statement on page 14-89
- Function Declaration and Definition on page 13-75 for information about creating a function in a PL/SQL block
- **Parameter Declaration** on page 13-99
- **Chapter 8, "Using PL/SQL Subprograms"**

**See Also:**
- *Oracle Database SQL Language Reference* for information about the `CALL` statement
- *Oracle Database Advanced Application Developer’s Guide* for information about restrictions on user-defined functions that are called from SQL statements and expressions
- *Oracle Database Advanced Application Developer’s Guide* for more information about call specifications
CREATE PACKAGE Statement

The CREATE PACKAGE statement creates or replaces the specification for a stored package, which is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The package specification declares these objects. The package body, specified subsequently, defines these objects.

Prerequisites

To create or replace a package in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a package in another user’s schema, you must have the CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE statement inside an the database precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

Syntax

create_package ::= 

\[ CREATE \ OR \ REPLACE \ PACKAGE \ [ \ invoker_rights_clause \ IS \ AS ] \ [ \ item_list_1 \ ] \ END \ package_name \]

(invoker_rights_clause ::= on page 14-39, item_list_1 ::= on page 13-8)

invoker_rights_clause ::= 

\[ AUTHID \{ \ CURRENT_USER \ | \ DEFINER \} \]

Keyword and Parameter Descriptions

OR REPLACE

Specify OR REPLACE to re-create the package specification if it already exists. Use this clause to change the specification of an existing package without dropping, re-creating, and regranting object privileges previously granted on the package. If you change a package specification, then the database recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, then the database marks the indexes DISABLED.

schema

Specify the schema to contain the package. If you omit schema, then the database creates the package in your own schema.
**item_list_1**
Declares package elements. If an item in item_list_1 is a pragma, it must one of the following:

- **RESTRICT_REFERENCES Pragma** on page 13-107
- **SERIALLY_REUSABLE Pragma** on page 13-120

**package_name**
A package stored in the database. For naming conventions, see Identifiers on page 2-4.

**invoker_rights_clause**
Specifies the AUTHID property of the member functions and procedures of the object type. For information about the AUTHID property, see "Using Invoker's Rights or Definer's Rights (AUTHID Clause)" on page 8-18.

**AUTHID CURRENT_USER**
Specify CURRENT_USER to indicate that the package executes with the privileges of CURRENT_USER. This clause creates an **invoker's rights package**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the package resides.

**AUTHID DEFINER**
Specify DEFINER to indicate that the package executes with the privileges of the owner of the schema in which the package resides and that external names resolve in the schema where the package resides. This is the default and creates a **definer's rights package**.

See Also: Using Invoker's Rights or Definer's Rights (AUTHID Clause) on page 8-18 for more information about invoker’s rights and definer’s rights

**item_list_1**
Declares a list of items. For syntax, see **Block** on page 13-8.

If an item in item_list_1 is a pragma, it must one of the following:

- **RESTRICT_REFERENCES Pragma** on page 13-107
- **SERIALLY_REUSABLE Pragma** on page 13-120

**Example**

**Creating a Package: Example** The following statement creates the specification of the emp_mgmt package.

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
    FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2,
                   manager_id NUMBER, salary NUMBER,
                   commission_pct NUMBER, department_id NUMBER)
               RETURN NUMBER;
    FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
               RETURN NUMBER;
    PROCEDURE remove_emp(employee_id NUMBER);
    PROCEDURE remove_dept(department_id NUMBER);
```

The specification for the `emp_mgmt` package declares the following public program objects:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`
- The exceptions `no_comm` and `no_sal`

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that call any of these public procedures or functions or raise any of the public exceptions of the package.

Before you can call this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a `CREATE PACKAGE BODY` statement that creates the body of the `emp_mgmt` package, see `CREATE PACKAGE BODY Statement` on page 14-42.

Related Topics

- `ALTER PACKAGE Statement` on page 14-6
- `CREATE PACKAGE Statement` on page 14-39
- `CREATE PACKAGE BODY Statement` on page 14-42
- `DROP PACKAGE Statement` on page 14-91
- `Function Declaration and Definition` on page 13-75
- `Procedure Declaration and Definition` on page 13-101
- Chapter 10, "Using PL/SQL Packages"
CREATE PACKAGE BODY Statement

The CREATE PACKAGE BODY statement creates or replaces the body of a stored package, which is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The package body defines these objects. The package specification, defined in an earlier CREATE PACKAGE statement, declares these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

Prerequisites

To create or replace a package in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a package in another user's schema, you must have the CREATE ANY PROCEDURE system privilege. In both cases, the package body must be created in the same schema as the package.

To embed a CREATE PACKAGE BODY statement inside an the database precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

Syntax

create_package_body ::=  
CREATE OR REPLACE PACKAGE BODY schema.package_name IS AS body END package_name;

(declare_section ::= on page 13-8, body ::= on page 13-11)

Keyword and Parameter Descriptions

OR REPLACE
Specify OR REPLACE to re-create the package body if it already exists. Use this clause to change the body of an existing package without dropping, re-creating, and regranting object privileges previously granted on it. If you change a package body, then the database recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

schema
Specify the schema to contain the package. If you omit schema, then the database creates the package in your current schema.

package_name
Specify the name of the package to be created.
**declare_section**
Declares package objects.

**body**
Defines package objects.

**Examples**

**Creating a Package Body: Example**  
This statement creates the body of the emp_mgmt package created in Creating a Package: Example on page 14-40.

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
  tot_emps NUMBER;
  tot_depts NUMBER;
  FUNCTION hire
    (last_name VARCHAR2, job_id VARCHAR2,
     manager_id NUMBER, salary NUMBER,
     commission_pct NUMBER, department_id NUMBER)
  RETURN NUMBER IS new_empno NUMBER;
  BEGIN
    SELECT employees_seq.NEXTVAL
    INTO new_empno
    FROM DUAL;
    INSERT INTO employees
      VALUES (new_empno, 'First', 'Last','first.example@oracle.com',
      '(415)555-0100','18-JUN-02','IT_PROG',90000000,00,
      100,110);
    tot_emps := tot_emps + 1;
    RETURN(new_empno);
  END;
  FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
  RETURN NUMBER IS
  new_deptno NUMBER;
  BEGIN
    SELECT departments_seq.NEXTVAL
    INTO new_deptno
    FROM dual;
    INSERT INTO departments
      VALUES (new_deptno, 'department name', 100, 1700);
    tot_depts := tot_depts + 1;
    RETURN(new_deptno);
  END;
  PROCEDURE remove_emp (employee_id NUMBER) IS
  BEGIN
    DELETE FROM employees
    WHERE employees.employee_id = remove_emp.employee_id;
    tot_emps := tot_emps - 1;
  END;
  PROCEDURE remove_dept(department_id NUMBER) IS
  BEGIN
    DELETE FROM departments
    WHERE departments.department_id = remove_dept.department_id;
    tot_depts := tot_depts - 1;
  END;
  PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER) IS
  curr_sal NUMBER;
  BEGIN
    SELECT salary INTO curr_sal FROM employees
  ```
WHERE employees.employee_id = increase_sal.employee_id;
IF curr_sal IS NULL
    THEN RAISE no_sal;
ELSE
    UPDATE employees
    SET salary = salary + salary_incr
    WHERE employee_id = employee_id;
END IF;
END;

PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER) IS
    curr_comm NUMBER;
    BEGIN
        SELECT commission_pct
        INTO curr_comm
        FROM employees
        WHERE employees.employee_id = increase_comm.employee_id;
        IF curr_comm IS NULL
            THEN RAISE no_comm;
        ELSE
            UPDATE employees
            SET commission_pct = commission_pct + comm_incr;
        END IF;
    END;
END emp_mgmt;
/

The package body defines the public program objects declared in the package specification:

- The functions hire and create_dept
- The procedures remove_emp, remove_dept, increase_sal, and increase_comm

These objects are declared in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure increase_all_comms separate from the emp_mgmt package that calls the increase_comm procedure.

These objects are defined in the package body, so you can change their definitions without causing the database to invalidate dependent schema objects. For example, if you subsequently change the definition of hire, then the database need not recompile increase_all_comms before executing it.

The package body in this example also declares private program objects, the variables tot_emps and tot_depts. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable tot_depts. However, the function create_dept is part of the package, so create_dept can change the value of tot_depts.

Related Topics

- CREATE PACKAGE Statement on page 14-39
- Function Declaration and Definition on page 13-75
- Procedure Declaration and Definition on page 13-101
- Chapter 10, “Using PL/SQL Packages”
CREATE PROCEDURE Statement

The CREATE PROCEDURE statement creates or replaces a standalone stored procedure or a call specification.

A standalone stored procedure is a procedure (a subprogram that performs a specific action) that is stored in the database.

---

**Note:** A standalone stored procedure that you create with the CREATE PROCEDURE statement is different from a procedure that you declare and define in a PL/SQL block or package. For information about the latter, see Procedure Declaration and Definition on page 13-101.

---

A call specification declares a Java method or a third-generation language (3GL) routine so that it can be called from PL/SQL. You can also use the SQL CALL statement to call such a method or routine. The call specification tells the database which Java method, or which named procedure in which shared library, to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

**Prerequisites**

To create or replace a standalone stored procedure in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a standalone stored procedure in another user’s schema, you must have the CREATE ANY PROCEDURE system privilege.

To invoke a call specification, you may need additional privileges, for example, the EXECUTE object privilege on the C library for a C call specification.

To embed a CREATE PROCEDURE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

**See Also:** For more information about such prerequisites:

- Oracle Database Advanced Application Developer’s Guide
- Oracle Database Java Developer’s Guide
CREATE PROCEDURE Statement

Syntax

create_procedure ::= 

ORSPECIFICATION (parameter_declaration ::= on page 13-99, declare_section ::= on page 13-8, body ::= on page 13-11)

invoker_rights_clause ::= 

AUTHID CURRENT_USER DEFINER

call_spec ::= 

LANGUAGE Java_declaration C_declaration

Java_declaration ::= 

JAVA NAME string

C_declaration ::= 

C NAME name LIBRARY lib_name AGENT IN argument WITH CONTEXT PARAMETERS parameter

Keyword and Parameter Descriptions

OR REPLACE

Specify OR REPLACE to re-create the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping, re-creating, and
regranting object privileges previously granted on it. If you redefine a procedure, then
the database recompiles it.

Users who had previously been granted privileges on a redefined procedure can still
access the procedure without being regranted the privileges.

If any function-based indexes depend on the package, then the database marks the
indexes DISABLED.

**schema**

Specify the schema to contain the procedure. If you omit `schema`, then the database
creates the procedure in your current schema.

**procedure_name**

Specify the name of the procedure to be created.

**invoker_rights_clause**

Specifies the AUTHID property of the member functions and procedures of the object
type. For information about the AUTHID property, see "Using Invoker's Rights or
Definer's Rights (AUTHID Clause)" on page 8-18.

**AUTHID CURRENT_USER**

Specify CURRENT_USER to indicate that the procedure executes with the privileges of
CURRENT_USER. This clause creates an **invoker's rights procedure**.

This clause also specifies that external names in queries, DML operations, and
dynamic SQL statements resolve in the schema of CURRENT_USER. External names in
all other statements resolve in the schema in which the procedure resides.

**AUTHID DEFINER**

Specify DEFINER to indicate that the procedure executes with the privileges of the
owner of the schema in which the procedure resides, and that external names resolve
in the schema where the procedure resides. This is the default and creates a **definer's
rights procedure**.

**body**

The required executable part of the procedure and, optionally, the exception-handling
part of the procedure.

**declare_section**

The optional declarative part of the procedure. Declarations are local to the procedure,
can be referenced in `body`, and cease to exist when the procedure completes execution.

**call_spec**

Use the `call_spec` to map a Java or C method name, parameter types, and return
type to their SQL counterparts.

In the `Java_declaration, string` identifies the Java implementation of the
method.
The EXTERNAL clause is an alternative way of declaring a C method. In most cases, Oracle recommends that you use the LANGUAGE C syntax. However, EXTERNAL is required if a default argument is used as one of the parameters or if one of the parameters uses a PL/SQL data type that must be mapped (for example, Boolean). EXTERNAL causes the PL/SQL layer to be loaded so that the parameters can be properly evaluated.

Examples

Creating a Procedure: Example  The following statement creates the procedure remove_emp in the schema hr.

```sql
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
  tot_emps NUMBER;
  BEGIN
    DELETE FROM employees
    WHERE employees.employee_id = remove_emp.employee_id;
    tot_emps := tot_emps - 1;
  END;
/
```

The remove_emp procedure removes a specified employee. When you call the procedure, you must specify the employee_id of the employee to be removed.

The procedure uses a DELETE statement to remove from the employees table the row of employee_id.

See Also:  Creating a Package Body: Example on page 14-43 to see how to incorporate this procedure into a package

In the following example, external procedure c_find_root expects a pointer as a parameter. Procedure find_root passes the parameter by reference using the BY REFERENCE phrase.

```sql
CREATE PROCEDURE find_root
  ( x IN REAL )
IS LANGUAGE C
  NAME c_find_root
  LIBRARY c_utils
  PARAMETERS ( x BY REFERENCE );
```

Related Topics

- ALTER PROCEDURE Statement on page 14-9
- CREATE FUNCTION Statement on page 14-29
- DROP PROCEDURE Statement on page 14-93
- Parameter Declaration on page 13-99
- Procedure Declaration and Definition on page 13-101
- Chapter 8, "Using PL/SQL Subprograms"

See Also:

- *Oracle Database SQL Language Reference* for information about the *CALL* statement
- *Oracle Database Advanced Application Developer’s Guide* for more information about call specifications
CREATE TRIGGER Statement

The CREATE TRIGGER statement creates or replaces a database trigger, which is either of the following:

- A stored PL/SQL block associated with a table, a schema, or the database
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

The database automatically executes a trigger when specified conditions occur.

Order of Trigger Firing If two or more triggers with different timing points (BEFORE, AFTER, INSTEAD OF) are defined for the same statement on the same table, then they fire in the following order:

- All BEFORE statement triggers
- All BEFORE row triggers
- All AFTER row triggers
- All AFTER statement triggers

If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order you intend.

If two or more triggers are defined with the same timing point, and the order in which they fire is important, then you can control the firing order using the FOLLOWS clause (see FOLLOWS on page 14-61).

If multiple compound triggers are specified on a table, then all BEFORE statement sections will be executed at the BEFORE statement timing point, BEFORE row sections will be executed at the BEFORE row timing point, and so forth. If trigger execution order has been specified using the FOLLOWS clause, then order of execution of compound trigger sections will be determined by the FOLLOWS clause. If FOLLOWS is specified only for some triggers but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the FOLLOWS clause.

Prerequisites

- To create a trigger in your own schema on a table in your own schema or on your own schema (SCHEMA), you must have the CREATE TRIGGER system privilege.
- To create a trigger in any schema on a table in any schema, or on another user's schema (schema.SCHEMA), you must have the CREATE ANY TRIGGER system privilege.
- In addition to the preceding privileges, to create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.
CREATE TRIGGER Statement

Syntax

create_trigger ::= 

CREATE [OR REPLACE] TRIGGER schema.trigger 

NON_DML_TRIGGER ::= 

CREATE OR REPLACE TRIGGER 

SIMPLE_DML_TRIGGER ::= 

COMPOUND_DML_TRIGGER ::= 

NON_DML_TRIGGER ::= 

FOLLOWS schema.trigger, 

ENABLE | DISABLE WHEN (condition) 

TRIGGER_BODY ::= 

BEFORE | AFTER | INSTEAD OF 

DML_EVENT_CLAUSE ::= 

REFERENCING_CLAUSE ::= 

FOR EACH ROW 

BEFORE | AFTER 

DDL_EVENT | DATABASE_EVENT 

PLSQL_BLOCK ::= 

COMPOUND_TRIGGER_BLOCK ::= 

CALL ROUTINE_CLAUSE 

( non_dml_trigger ::= on page 14-53, trigger_body ::= on page 14-53)
CREATE TRIGGER Statement

(dml_event_clause ::= on page 13-8, compound_trigger_block ::= on page 14-54, CALL routine_clause in Oracle Database SQL Language Reference)

```
dml_event_clause ::= 

REFERENCING 

referencing_clause ::= 

compound_trigger_block ::= 

timing_point_section ::= 
```
Keyword and Parameter Descriptions

**OR REPLACE**
Specify OR REPLACE to re-create the trigger if it already exists. Use this clause to change the definition of an existing trigger without first dropping it.

**schema**
Specify the schema to contain the trigger. If you omit schema, then the database creates the trigger in your own schema.

**trigger**
Specify the name of the trigger to be created.

If a trigger produces compilation errors, then it is still created, but it fails on execution. This means it effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

---

**Note:** If you create a trigger on a base table of a materialized view, then you must ensure that the trigger does not fire during a refresh of the materialized view. During refresh, the `DBMS_MVIEW` procedure `I_AM_A_REFRESH` returns `TRUE`.

---

**simple_dml_trigger**
Use this clause to define a single trigger on a DML event.

**BEFORE**
Specify BEFORE to cause the database to fire the trigger before executing the triggering event. For row triggers, the trigger is fired before each affected row is changed.

**Restrictions on BEFORE Triggers**
BEFORE triggers are subject to the following restrictions:

- You cannot specify a BEFORE trigger on a view.
- In a BEFORE statement trigger, or in BEFORE statement section of a compound trigger, you cannot specify either :NEW or :OLD. A BEFORE row trigger or a BEFORE row section of a compound trigger can read and write into the :OLD or :NEW fields.
**AFTER**
Specify **AFTER** to cause the database to fire the trigger after executing the triggering event. For row triggers, the trigger is fired after each affected row is changed.

**Restrictions on AFTER Triggers**  
**AFTER** triggers are subject to the following restrictions:
- You cannot specify an **AFTER** trigger on a view.
- In an **AFTER** statement trigger or in **AFTER** statement section of a compound trigger, you cannot specify either :NEW or :OLD. An **AFTER** row trigger or **AFTER** row section of a compound trigger can only read but not write into the :OLD or :NEW fields.

**Note:** When you create a materialized view log for a table, the database implicitly creates an **AFTER ROW** trigger on the table. This trigger inserts a row into the materialized view log whenever an INSERT, UPDATE, or DELETE statement modifies data in the master table. You cannot control the order in which multiple row triggers fire. Therefore, you should not write triggers intended to affect the content of the materialized view.

**See Also:** *Oracle Database SQL Language Reference* for more information about materialized view logs

**INSTEAD OF**
Specify **INSTEAD OF** to cause the database to fire the trigger instead of executing the triggering event. You can achieve the same effect when you specify an **INSTEAD OF ROW** section in a compound trigger.

**Note:** the database fine-grained access control lets you define row-level security policies on views. These policies enforce specified rules in response to DML operations. If an **INSTEAD OF** trigger is also defined on the view, then the database will not enforce the row-level security policies, because the database fires the **INSTEAD OF** trigger instead of executing the DML on the view.

- **INSTEAD OF** triggers are valid for DML events on any views. They are not valid for DDL or database events, and you cannot specify an **INSTEAD OF** trigger on a table.
- You can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW value.
- If a view is inherently updatable and has **INSTEAD OF** triggers, then the triggers take preference. The database fires the triggers instead of performing DML on the view.
- If the view belongs to a hierarchy, then the trigger is not inherited by subviews.

**See Also:** Creating an **INSTEAD OF** Trigger: Example on page 14-63
**dm1_event_clause**

The `dm1_event_clause` lets you specify one of three DML statements that can cause the trigger to fire. The database fires the trigger in the existing user transaction.

You cannot specify the `MERGE` keyword in the `dm1_event_clause`. If you want a trigger to fire in relation to a `MERGE` operation, then you must create triggers on the `INSERT` and `UPDATE` operations to which the `MERGE` operation decomposes.

**See Also:** Creating a DML Trigger: Examples on page 14-62

**DELETE** Specify `DELETE` if you want the database to fire the trigger whenever a `DELETE` statement removes a row from the table or removes an element from a nested table.

**INSERT** Specify `INSERT` if you want the database to fire the trigger whenever an `INSERT` statement adds a row to a table or adds an element to a nested table.

**UPDATE** Specify `UPDATE` if you want the database to fire the trigger whenever an `UPDATE` statement changes a value in one of the columns specified after `OF`. If you omit `OF`, then the database fires the trigger whenever an `UPDATE` statement changes a value in any column of the table or nested table.

For an `UPDATE` trigger, you can specify object type, varray, and `REF` columns after `OF` to indicate that the trigger should be fired whenever an `UPDATE` statement changes a value in one of the columns. However, you cannot change the values of these columns in the body of the trigger itself.

---

**Note:** Using OCI functions or the `DBMS_LOB` package to update LOB values or LOB attributes of object columns does not cause the database to fire triggers defined on the table containing the columns or the attributes.

---

**Restrictions on Triggers on UPDATE Operations** The `UPDATE` clause is subject to the following restrictions:

- You cannot specify `UPDATE OF` for an `INSTEAD OF` trigger. The database fires `INSTEAD OF` triggers whenever an `UPDATE` changes a value in any column of the view.
- You cannot specify a nested table or LOB column in the `UPDATE OF` clause.

**See Also:** `AS subquery` clause of `CREATE VIEW` in Oracle Database SQL Language Reference for a list of constructs that prevent inserts, updates, or deletes on a view

Performing DML operations directly on nested table columns does not cause the database to fire triggers defined on the table containing the nested table column.

**ON table | view** The `ON` clause lets you determine the database object on which the trigger is to be created. Specify the `schema` and `table or view` name of one of the following on which the trigger is to be created:

- Table or view
- Object table or object view
- A column of nested-table type
If you omit *schema*, then the database assumes the table is in your own schema.

**Restriction on Schema**  You cannot create a trigger on a table in the schema *SYS*.

**NESTED TABLE Clause**  Specify the *nested_table_column* of a view upon which the trigger is being defined. Such a trigger will fire only if the DML operates on the elements of the nested table.

**Restriction on Triggers on Nested Tables**  You can specify NESTED TABLE only for INSTEAD OF triggers.

**referencing_clause**

The *referencing_clause* lets you specify correlation names. You can use correlation names in the trigger body and *WHEN* condition of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, then use this clause to specify different correlation names to avoid confusion between the table name and the correlation name.

- If the trigger is defined on a nested table, then OLD and NEW refer to the row of the nested table, and PARENT refers to the current row of the parent table.
- If the trigger is defined on an object table or view, then OLD and NEW refer to object instances.

**Restriction on the referencing_clause**  The *referencing_clause* is not valid with INSTEAD OF triggers on CREATE DDL events.

**FOR EACH ROW**

Specify FOR EACH ROW to designate the trigger as a row trigger. The database fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the *WHEN* condition.

Except for INSTEAD OF triggers, if you omit this clause, then the trigger is a statement trigger. The database fires a statement trigger only once when the triggering statement is issued if the optional trigger constraint is met.

INSTEAD OF trigger statements are implicitly activated for each row.

**Restriction on Row Triggers**  This clause is valid only for simple DML triggers, not for compound DML triggers or for DDL or database event triggers.

**compound_dml_trigger**

Use this clause to define a compound trigger on a DML event. The body of a COMPOUND trigger can have up to four sections, so that you can specify a before statement, before row, after row, or after statement operation in one trigger.

The *dml_event_clause* and the *referencing_clause* have the same semantics for compound DML triggers as for simple DML triggers.

**Restriction on Compound Triggers**  You cannot specify the FOR EACH ROW clause for a compound trigger.

**See Also:**  Compound Trigger Restrictions on page 9-15 for additional restrictions
non_dml_trigger

Use this clause to define a single trigger on a DDL or database event.

ddl_event

Specify one or more types of DDL statements that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can create BEFORE and AFTER triggers for these events. The database fires the trigger in the existing user transaction.

Restriction on Triggers on DDL Events  You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

See Also: Creating a DDL Trigger: Example on page 14-63

The following ddl_event values are valid:

ALTER  Specify ALTER to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary. The trigger will not be fired by an ALTER DATABASE statement.

ANALYZE  Specify ANALYZE to fire the trigger whenever the database collects or deletes statistics or validates the structure of a database object.

ASSOCIATE STATISTICS  Specify ASSOCIATE STATISTICS to fire the trigger whenever the database associates a statistics type with a database object.

AUDIT  Specify AUDIT to fire the trigger whenever the database tracks the occurrence of a SQL statement or tracks operations on a schema object.

COMMENT  Specify COMMENT to fire the trigger whenever a comment on a database object is added to the data dictionary.

CREATE  Specify CREATE to fire the trigger whenever a CREATE statement adds a new database object to the data dictionary. The trigger will not be fired by a CREATE DATABASE or CREATE CONTROLFILE statement.

DISASSOCIATE STATISTICS  Specify DISASSOCIATE STATISTICS to fire the trigger whenever the database disassociates a statistics type from a database object.

DROP  Specify DROP to fire the trigger whenever a DROP statement removes a database object from the data dictionary.

GRANT  Specify GRANT to fire the trigger whenever a user grants system privileges or roles or object privileges to another user or to a role.

NOAUDIT  Specify NOAUDIT to fire the trigger whenever a NOAUDIT statement instructs the database to stop tracking a SQL statement or operations on a schema object.

RENAME  Specify RENAME to fire the trigger whenever a RENAME statement changes the name of a database object.
CREATE TRIGGER Statement

REVOKE     Specify REVOKE to fire the trigger whenever a REVOKE statement removes system privileges or roles or object privileges from a user or role.

TRUNCATE   Specify TRUNCATE to fire the trigger whenever a TRUNCATE statement removes the rows from a table or cluster and resets its storage characteristics.

DDL        Specify DDL to fire the trigger whenever any of the preceding DDL statements is issued.

database_event
Specify one or more particular states of the database that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. For each of these triggering events, the database opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction).

    See Also:
    ■ Creating a Database Event Trigger: Example on page 14-63
    ■ Responding to Database Events Through Triggers on page 9-45 for more information about responding to database events through triggers

Each database event is valid in either a BEFORE trigger or an AFTER trigger, but not both. The following database_event values are valid:

    AFTER STARTUP     Specify AFTER STARTUP to fire the trigger whenever the database is opened. This event is valid only with DATABASE, not with SCHEMA.

    BEFORE SHUTDOWN   Specify BEFORE SHUTDOWN to fire the trigger whenever an instance of the database is shut down. This event is valid only with DATABASE, not with SCHEMA.

    AFTER DB_ROLECHANGE In a Data Guard configuration, specify AFTER DB_ROLECHANGE to fire the trigger whenever a role change occurs from standby to primary or from primary to standby. This event is valid only with DATABASE, not with SCHEMA.

    AFTER LOGON       Specify AFTER LOGON to fire the trigger whenever a client application logs onto the database.

    BEFORE LOGOFF     Specify BEFORE LOGOFF to fire the trigger whenever a client application logs off the database.

    AFTER SERVERERROR Specify AFTER SERVERERROR to fire the trigger whenever a server error message is logged.

    The following errors do not cause a SERVERERROR trigger to fire:
    ■ ORA-01403: no data found
    ■ ORA-01422: exact fetch returns more than requested number of rows
    ■ ORA-01423: error encountered while checking for extra rows in exact fetch
    ■ ORA-01034: ORACLE not available
CREATE TRIGGER Statement

- ORA-04030: out of process memory when trying to allocate string bytes (string, string)

AFTER SUSPEND  Specify SUSPEND to fire the trigger whenever a server error causes a transaction to be suspended.

See Also: Doing Independent Units of Work with Autonomous Transactions on page 6-40 for information about autonomous transactions

DATABASE  Specify DATABASE to define the trigger on the entire database. The trigger fires whenever any database user initiates the triggering event.

SCHEMA  Specify SCHEMA to define the trigger on the current schema. The trigger fires whenever any user connected as schema initiates the triggering event.

See Also: Creating a SCHEMA Trigger: Example on page 14-64

FOLLOWS
This clause lets you specify the relative firing order of triggers of the same type. Use FOLLOWS to indicate that the trigger being created should fire after the specified triggers.

The specified triggers must already exist, they must be defined on the same table as the trigger being created, and they must have been successfully compiled. They need not be enabled.

You can specify FOLLOWS in the definition of a simple trigger with a compound trigger target, or in the definition of a compound trigger with a simple trigger target. In these cases, the FOLLOWS keyword applies only to the section of the compound trigger with the same timing point as the sample trigger. If the compound trigger has no such timing point, then FOLLOWS is quietly ignored.

See Also: Order of Trigger Firing on page 14-51 for more information about the order in which the database fires triggers

ENABLE | DISABLE
Use this clause to create the trigger in an enabled or disabled state. Creating a trigger in a disabled state lets you ensure that the trigger compiles without errors before you put into actual use.

Specify DISABLE to create the trigger in disabled form. You can subsequently issue an ALTER TRIGGER ... ENABLE or ALTER TABLE ... ENABLE ALL TRIGGERS statement to enable the trigger. If you omit this clause, then the trigger is enabled when it is created.

See Also:
- ALTER TRIGGER Statement on page 14-12 for information about is ENABLE clause
- Oracle Database SQL Language Reference for information about using CREATE TABLE ... ENABLE ALL TRIGGERS

WHEN Clause
Specify the trigger condition, which is a SQL condition that must be satisfied for the database to fire the trigger. This condition must contain correlation names and cannot contain a query.
The NEW and OLD keywords, when specified in the WHEN clause, are not considered bind variables, so are not preceded by a colon (:). However, you must precede NEW and OLD with a colon in all references other than the WHEN clause.

See Also:
- Oracle Database SQL Language Reference for the syntax description of condition
- Calling a Procedure in a Trigger Body: Example on page 14-63

Restrictions on Trigger Conditions Trigger conditions are subject to the following restrictions:

- If you specify this clause for a DML event trigger, then you must also specify FOR EACH ROW. the database evaluates this condition for each row affected by the triggering statement.
- You cannot specify trigger conditions for INSTEAD OF trigger statements.
- You can reference object columns or their attributes, or varray, nested table, or LOB columns. You cannot invoke PL/SQL functions or methods in the trigger condition.

trigger_body
Specify the PL/SQL block, PL/SQL compound trigger block, or call procedure that the database executes to fire the trigger.

compound_trigger_block
Timing point sections can be in any order, but no timing point section can be repeated. The declare_section of a compound trigger block cannot include PRAGMA AUTONOMOUS_TRANSACTION.

Examples

Creating a DML Trigger: Examples This example shows the basic syntax for a BEFORE statement trigger. You would write such a trigger to place restrictions on DML statements issued on a table, for example, when such statements could be issued.

```
CREATE TRIGGER schema.trigger_name
    BEFORE DELETE OR INSERT OR UPDATE
    ON schema.table_name
    pl/sql_block
```

the database fires such a trigger whenever a DML statement affects the table. This trigger is a BEFORE statement trigger, so the database fires it once before executing the triggering statement.

The next example shows a partial BEFORE row trigger. The PL/SQL block might specify, for example, that an employee's salary must fall within the established salary range for the employee's job:

```
CREATE TRIGGER hr.salary_check
    BEFORE INSERT OR UPDATE OF salary, job_id ON hr.employees
    FOR EACH ROW
    WHEN (new.job_id <> 'AD_VP')
    pl/sql_block
```
the database fires this trigger whenever one of the following statements is issued:

- An `INSERT` statement that adds rows to the `employees` table
- An `UPDATE` statement that changes values of the `salary` or `job_id` columns of the `employees` table

`salary_check` is a `BEFORE` row trigger, so the database fires it before changing each row that is updated by the `UPDATE` statement or before adding each row that is inserted by the `INSERT` statement.

`salary_check` has a trigger condition that prevents it from checking the salary of the administrative vice president (`AD_VP`).

**Creating a DDL Trigger: Example**  
This example creates an `AFTER` statement trigger on any DDL statement `CREATE`. Such a trigger can be used to audit the creation of new data dictionary objects in your schema.

```
CREATE TRIGGER audit_db_object AFTER CREATE
  ON SCHEMA
  pl/sql_block
```

**Calling a Procedure in a Trigger Body: Example**  
You could create the `salary_check` trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume you have defined a procedure `check_sal` in the `hr` schema, which verifies that an employee's salary is in an appropriate range. Then you could create the trigger `salary_check` as follows:

```
CREATE TRIGGER salary_check
  BEFORE INSERT OR UPDATE OF salary, job_id ON employees
  FOR EACH ROW
  WHEN (new.job_id <> 'AD_VP')
  CALL check_sal(:new.job_id, :new.salary, :new.last_name)
```

The procedure `check_sal` could be implemented in PL/SQL, C, or Java. Also, you can specify :OLD values in the `CALL` clause instead of :NEW values.

**Creating a Database Event Trigger: Example**  
This example shows the basic syntax for a trigger to log all errors. The hypothetical PL/SQL block does some special processing for a particular error (invalid logon, error number 1017). This trigger is an `AFTER` statement trigger, so it is fired after an unsuccessful statement execution, such as unsuccessful logon.

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
BEGIN
  IF (IS_SERVERERROR (1017)) THEN
    <special processing of logon error>
  ELSE
    <log error number>
  END IF;
END;
```

**Creating an INSTEAD OF Trigger: Example**  
In this example, an `oe.order_info` view is created to display information about customers and their orders:

```
CREATE VIEW order_info AS
  SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
         o.order_id, o.order_date, o.order_status
  FROM customers c, orders o
WHERE c.customer_id = o.customer_id;
```
Normally this view would not be updatable, because the primary key of the orders table (order_id) is not unique in the result set of the join view. To make this view updatable, create an INSTEAD OF trigger on the view to process INSERT statements directed to the view.

```sql
CREATE OR REPLACE TRIGGER order_info_insert
    INSTEAD OF INSERT ON order_info
BEGIN
    DECLARE
duplicate_info EXCEPTION;
    PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
    BEGIN
        INSERT INTO customers
            (customer_id, cust_last_name, cust_first_name)
        VALUES (:new.customer_id,
            :new.cust_last_name,
            :new.cust_first_name);
        INSERT INTO orders (order_id, order_date, customer_id)
        VALUES (:new.order_id,
            :new.order_date,
            :new.customer_id);
        EXCEPTION
        WHEN duplicate_info THEN
            RAISE_APPLICATION_ERROR (
                num=> -20107,
                msg=> 'Duplicate customer or order ID');
    END order_info_insert;
/
```

You can now insert into both base tables through the view (as long as all NOT NULL columns receive values):

```sql
INSERT INTO order_info VALUES
    (999, 'Smith', 'John', 2500, '13-MAR-2001', 0);
```

For more information about INSTEAD OF triggers, see Modifying Complex Views (INSTEAD OF Triggers) on page 9-8.

**Creating a SCHEMA Trigger: Example** The following example creates a BEFORE statement trigger on the sample schema hr. When a user connected as hr attempts to drop a database object, the database fires the trigger before dropping the object:

```sql
CREATE OR REPLACE TRIGGER drop_trigger
    BEFORE DROP ON hr.SCHEMA
BEGIN
    RAISE_APPLICATION_ERROR (
        num => -20000,
        msg => 'Cannot drop object');
END;
/
```

**Related Topics**

- ALTER TRIGGER Statement on page 14-12
- DROP TRIGGER Statement on page 14-94
- Chapter 9, "Using Triggers"
CREATE TYPE Statement

The **CREATE TYPE** statement creates or replaces the specification of an **object type**, a **SQLJ object type**, a named varying array (**varray**), a **nested table type**, or an **incomplete object type**. You create object types with the **CREATE TYPE** and the **CREATE TYPE BODY** statements. The **CREATE TYPE** statement specifies the name of the object type, its attributes, methods, and other properties. The **CREATE TYPE BODY** statement contains the code for the methods that implement the type.

**Notes:**
- If you create an object type for which the type specification declares only attributes but no methods, then you need not specify a type body.
- If you create a SQLJ object type, then you cannot specify a type body. The implementation of the type is specified as a Java class.

An **incomplete type** is a type created by a forward type definition. It is called "incomplete" because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

**Note:** A standalone stored type that you create with the **CREATE TYPE** statement is different from a type that you define in a PL/SQL block or package. For information about the latter, see Collection on page 13-22.

With the **CREATE TYPE** statement, you can create nested table and **varray** types, but not **associative arrays**. In a PL/SQL block or package, you can define all three collection types.

**Prerequisites**

To create a type in your own schema, you must have the **CREATE TYPE** system privilege. To create a type in another user’s schema, you must have the **CREATE ANY TYPE** system privilege. You can acquire these privileges explicitly or be granted them through a role.

To create a subtype, you must have the **UNDER ANY TYPE** system privilege or the **UNDER object privilege** on the supertype.

The owner of the type must be explicitly granted the **EXECUTE object privilege** in order to access all other types referenced within the definition of the type, or the type owner must be granted the **EXECUTE ANY TYPE** system privilege. The owner cannot obtain these privileges through roles.

If the type owner intends to grant other users access to the type, then the owner must be granted the **EXECUTE object privilege** on the referenced types with the **GRANT OPTION** or the **EXECUTE ANY TYPE** system privilege with the **ADMIN OPTION**. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.
Syntax

create_type ::= 

(object_type_def ::= on page 13-23, nested_table_type_def ::= on page 13-23)

object_type ::= 

(invoker_rights_clause ::= on page 14-67)

invoker_rights_clause ::= 

(sqlj_object_type ::= on page 14-67, sqlj_object_type_attr ::= on page 14-67)

sqlj_object_type ::= 

(sqlj_object_type_attr ::= on page 14-67)

sqlj_object_type_attr ::= 


CREATE TYPE Statement

**element_spec ::=**

- inheritance_clauses
  - subprogram_spec
  - constructor_spec
  - map_order_function_spec
- pragma_clause

(constructor_spec ::= on page 14-68, map_order_function_spec ::= on page 14-69, pragma_clause ::= on page 14-69)

**inheritance_clauses ::=**

- NOT
- OVERRIDEING
- FINAL
- INSTANTIABLE

**subprogram_spec ::=**

- MEMBER
- STATIC
- procedure_spec
- function_spec

**procedure_spec ::=**

- PROCEDURE
- procedure_name
- parameter
datatype
- IS
- call_spec

(call_spec ::= on page 14-70)

**function_spec ::=**

- FUNCTION
- name
- (parameter
datatype
- )
return_clause

(return_clause ::= on page 14-69)

**constructor_spec ::=**

- FINAL
- INSTANTIABLE
- CONSTRUCTOR
- FUNCTION
datatype
- SELF
- IN
- OUT
datatype
- parameter
datatype
- IS
- call_spec

- RETURN
- SELF
- AS
- RESULT

14-62 Oracle Database PL/SQL Language Reference
CREATE TYPE Statement

(SQL Statements for Stored PL/SQL Units 14-63)

(map_order_function_spec ::= )

(call_spec ::= on page 14-70)

return_clause ::= )

(call_spec ::= on page 14-70)

sqlj_object_type_sig ::= )

pragma_clause ::= )

(call_spec ::= on page 14-70)

Java_declaration ::= )
CREATE TYPE Statement

**C_declaration ::=**

```
C NAME name LIBRARY lib_name AGENT IN (argument) WITH CONTEXT PARAMETERS (parameter)
```

Keyword and Parameter Descriptions

**OR REPLACE**
Specify OR REPLACE to re-create the type if it already exists. Use this clause to change the definition of an existing type without first dropping it.

Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again.

If any function-based indexes depend on the type, then the database marks the indexes DISABLED.

**schema**
Specify the schema to contain the type. If you omit schema, then the database creates the type in your current schema.

**type_name**
Specify the name of an object type, a nested table type, or a varray type.

If creating the type results in compilation errors, then the database returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

The database implicitly defines a constructor method for each user-defined type that you create. A **constructor** is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the same as the name of the user-defined type. You can also create a user-defined constructor using the constructor_specs syntax.

The parameters of the object type constructor method are the data attributes of the object type. They occur in the same order as the attribute definition order for the object type. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

**object_type**
Use the object_type clause to create a user-defined object type. The variables that form the data structure are called attributes. The member subprograms that define the behavior of the object are called methods. The keywords AS OBJECT are required when creating an object type.

**See Also:** Object Type Examples on page 14-78
**OID 'object_identifier'**
The OID clause is useful for establishing type equivalence of identical objects in more than one database. (OID is short for Oracle Internet Directory.) See Oracle Database Object-Relational Developer’s Guide for information about this clause.

**invoker_rights_clause**
Specifies the AUTHID property of the member functions and procedures of the object type. For information about the AUTHID property, see "Using Invoker's Rights or Definer's Rights (AUTHID Clause)" on page 8-18.

**Restrictions on Invoker’s Rights**  This clause is subject to the following restrictions:
- You can specify this clause only for an object type, not for a nested table or varray type.
- You can specify this clause for clarity if you are creating a subtype. However, subtypes inherit the rights model of their supertypes, so you cannot specify a different value than was specified for the supertype.
- If the supertype was created with definer’s rights, then you must create the subtype in the same schema as the supertype.

**See Also:** Using Invoker's Rights or Definer's Rights (AUTHID Clause) on page 8-18 for more information about the AUTHID clause

**AS OBJECT Clause**
Specify AS OBJECT to create a top-level object type. Such object types are sometimes called root object types.

**UNDER Clause**
Specify UNDER supertype to create a subtype of an existing type. The existing supertype must be an object type. The subtype you create in this statement inherits the properties of its supertype. It must either override some of those properties or add new properties to distinguish it from the supertype.

**sqlj_object_type**
Specify this clause to create a SQLJ object type. In a SQLJ object type, you map a Java class to a SQL user-defined type. You can then define tables or columns on the SQLJ object type as you would with any other user-defined type.

You can map one Java class to multiple SQLJ object types. If there exists a subtype or supertype of a SQLJ object type, then it must also be a SQLJ object type. All types in the hierarchy must be SQLJ object types.

**java_ext_name**  Specify the name of the Java class. If the class exists, then it must be public. The Java external name, including the schema, will be validated.

Multiple SQLJ object types can be mapped to the same class. However:
- A subtype must be mapped to a class that is an immediate subclass of the class to which its supertype is mapped.
Two subtypes of a common supertype cannot be mapped to the same class.

**SQLData | CustomDatum | OraData** Choose the mechanism for creating the Java instance of the type. SQLData, CustomDatum, and OraData are the interfaces that determine which mechanism will be used.

**See Also:** *Oracle Database JDBC Developer’s Guide and Reference* for information about these three interfaces and *SQLJ Object Type Example* on page 14-78

code

element_spec

The `element_spec` lets you specify each attribute of the object type.

code

attribute

For `attribute`, specify the name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object. You must specify at least one attribute for each object type.

If you are creating a subtype, then the attribute name cannot be the same as any attribute or method name declared in the supertype chain.

code

datatype

For `datatype`, specify the database built-in data type or user-defined type of the attribute.

code

Restrictions on Attribute Data Types  Attribute data types are subject to the following restrictions:

- You cannot specify attributes of type `ROWID`, `LONG`, or `LONG RAW`.
- You cannot specify a data type of `UROWID` for a user-defined object type.
- If you specify an object of type `REF`, then the target object must have an object identifier.
- If you are creating a collection type for use as a nested table or varray column of a table, then you cannot specify attributes of type `ANYTYPE`, `ANYDATA`, or `ANYDATASET`.

**See Also:** Chapter 3, "PL/SQL Data Types," for a list of valid data types

code

sqlj_object_type_attr

This clause is valid only if you have specified the `sqlj_object_type` clause to map a Java class to a SQLJ object type. Specify the external name of the Java field that corresponds to the attribute of the SQLJ object type. The Java `field_name` must already exist in the class. You cannot map a Java `field_name` to more than one SQLJ object type attribute in the same type hierarchy.

This clause is optional when you create a SQLJ object type.

code

subprogram_spec

The `subprogram_spec` lets you associate a procedure subprogram with the object type.
MEMBER Clause
Specify a function or procedure subprogram associated with the object type that is referenced as an attribute. Typically, you invoke MEMBER methods in a selfish style, such as object_expression.method(). This class of method has an implicit first argument referenced as SELF in the method body, which represents the object on which the method has been invoked.

Restriction on Member Methods You cannot specify a MEMBER method if you are mapping a Java class to a SQLJ object type.

See Also: Creating a Member Method: Example on page 14-81

STATIC Clause
Specify a function or procedure subprogram associated with the object type. Unlike MEMBER methods, STATIC methods do not have any implicit parameters. You cannot reference SELF in their body. They are typically invoked as type_name.method().

Restrictions on Static Methods Static methods are subject to the following restrictions:

- You cannot map a MEMBER method in a Java class to a STATIC method in a SQLJ object type.
- For both MEMBER and STATIC methods, you must specify a corresponding method body in the object type body for each procedure or function specification.

See Also: Creating a Static Method: Example on page 14-81

[NOT] FINAL, [NOT] INSTANTIABLE
At the top level of the syntax, these clauses specify the inheritance attributes of the type.

Use the [NOT] FINAL clause to indicate whether any further subtypes can be created for this type:

- Specify FINAL if no further subtypes can be created for this type. This is the default.
- Specify NOT FINAL if further subtypes can be created under this type.

Use the [NOT] INSTANTIABLE clause to indicate whether any object instances of this type can be constructed:

- Specify INSTANTIABLE if object instances of this type can be constructed. This is the default.
- Specify NOT INSTANTIABLE if no default or user-defined constructor exists for this object type. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes, either inherited or specified in this statement.

inheritance_clauses
As part of the element_spec, the inheritance_clauses let you specify the relationship between supertypes and subtypes.

OVERRIDING This clause is valid only for MEMBER methods. Specify OVERRIDING to indicate that this method overrides a MEMBER method defined in the supertype. This
keyword is required if the method redefines a supertype method. NOT OVERRIDING is the default.

Restriction on OVERRIDING  The OVERRIDING clause is not valid for a STATIC method or for a SQLJ object type.

FINAL  Specify FINAL to indicate that this method cannot be overridden by any subtype of this type. The default is NOT FINAL.

NOT INSTANTIABLE  Specify NOT INSTANTIABLE if the type does not provide an implementation for this method. By default all methods are INSTANTIABLE.

Restriction on NOT INSTANTIABLE  If you specify NOT INSTANTIABLE, then you cannot specify FINAL or STATIC.

See Also:  constructor_spec on page 14-76

procedure_spec or function_spec

Use these clauses to specify the parameters and data types of the procedure or function. If this subprogram does not include the declaration of the procedure or function, then you must issue a corresponding CREATE TYPE BODY statement.

Restriction on Procedure and Function Specification  If you are creating a subtype, then the name of the procedure or function cannot be the same as the name of any attribute, whether inherited or not, declared in the supertype chain.

return_clause  The first form of the return_clause is valid only for a function. The syntax shown is an abbreviated form.

See Also:

- CREATE TYPE Statement on page 14-65 for more information about declaring object types
- Collection Method Call on page 13-27 for information about method invocation and methods
- CREATE PROCEDURE Statement on page 14-46 and CREATE FUNCTION Statement on page 14-29 for the full syntax with all possible clauses

sqlj_object_type_sig  Use this form of the return_clause if you intend to create SQLJ object type functions or procedures.

- If you are mapping a Java class to a SQLJ object type and you specify EXTERNAL NAME, then the value of the Java method returned must be compatible with the SQL returned value, and the Java method must be public. Also, the method signature (method name plus parameter types) must be unique within the type hierarchy.
- If you specify EXTERNAL VARIABLE NAME, then the type of the Java static field must be compatible with the return type.

call_spec

Specify the call specification that maps a Java or C method name, parameter types, and return type to their SQL counterparts. If all the member methods in the type have been
defined in this clause, then you need not issue a corresponding `CREATE TYPE BODY` statement.

The `Java declaration` string identifies the Java implementation of the method.

See Also:
- Oracle Database Java Developer’s Guide for an explanation of the parameters and semantics of the `Java declaration`
- Oracle Database Advanced Application Developer’s Guide for information about calling external procedures

`pragma_clause`

The `pragma_clause` lets you specify a compiler directive. The `PRAGMA RESTRICT_REFERENCES` compiler directive denies member functions read/write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

---

**Note:** Oracle recommends that you avoid using this clause unless you must do so for backward compatibility of your applications. This clause has been deprecated, because the database now runs purity checks at run time.

---

`method`  Specify the name of the `MEMBER` function or procedure to which the pragma is being applied.

`DEFAULT`  Specify `DEFAULT` if you want the database to apply the pragma to all methods in the type for which a pragma has not been explicitly specified.

`WNDS`  Specify `WNDS` to enforce the constraint writes no database state, which means that the method does not modify database tables.

`WNPS`  Specify `WNPS` to enforce the constraint writes no package state, which means that the method does not modify packaged variables.

`RNDS`  Specify `RNDS` to enforce the constraint reads no database state, which means that the method does not query database tables.

`RNPS`  Specify `RNPS` to enforce the constraint reads no package state, which means that the method does not reference package variables.

`TRUST`  Specify `TRUST` to indicate that the restrictions listed in the pragma are not actually to be enforced but are simply trusted to be true.

See Also: `RESTRICT_REFERENCES Pragma` on page 13-107 for more information about this pragma

`constructor_spec`

Use this clause to create a user-defined constructor, which is a function that returns an initialized instance of a user-defined object type. You can declare multiple constructors for a single object type, as long as the parameters of each constructor differ in number, order, or data type.

- User-defined constructor functions are always `FINAL` and `INSTANTIABLE`, so these keywords are optional.
- The parameter-passing mode of user-defined constructors is always SELF IN OUT. Therefore you need not specify this clause unless you want to do so for clarity.

- RETURN SELF AS RESULT specifies that the run-time type of the value returned by the constructor is the same as the run-time type of the SELF argument.

**See Also:** Oracle Database Object-Relational Developer’s Guide for more information about and examples of user-defined constructors and Constructor Example on page 14-81

**map_order_function_spec**

You can define either one MAP method or one ORDER method in a type specification, regardless of how many MEMBER or STATIC methods you define. If you declare either method, then you can compare object instances in SQL.

You cannot define either MAP or ORDER methods for subtypes. However, a subtype can override a MAP method if the supertype defines a nonfinal MAP method. A subtype cannot override an ORDER method at all.

You can specify either MAP or ORDER when mapping a Java class to a SQL type. However, the MAP or ORDER methods must map to MEMBER functions in the Java class. If neither a MAP nor an ORDER method is specified, then only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method must be specified to determine the equality of two object types.

Use MAP if you are performing extensive sorting or hash join operations on object instances. MAP is applied once to map the objects to scalar values, and then the database uses the scalars during sorting and merging. A MAP method is more efficient than an ORDER method, which must invoke the method for each object comparison. You must use a MAP method for hash joins. You cannot use an ORDER method because the hash mechanism hashes on the object value.

**See Also:** Oracle Database Object-Relational Developer’s Guide for more information about object value comparisons

**MAP MEMBER** This clause lets you specify a MAP member function that returns the relative position of a given instance in the ordering of all instances of the object. A MAP method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the MAP method is null, then the MAP method returns null and the method is not invoked.

An object specification can contain only one MAP method, which must be a function. The result type must be a predefined SQL scalar type, and the MAP method can have no arguments other than the implicit SELF argument.

**Note:** If type_name will be referenced in queries containing sorts (through an ORDER BY, GROUP BY, DISTINCT, or UNION clause) or containing joins, and you want those queries to be parallelized, then you must specify a MAP member function.
CREATE TYPE Statement

SQL Statements for Stored PL/SQL Units

A subtype cannot define a new MAP method. However it can override an inherited MAP method.

**ORDER MEMBER** This clause lets you specify an ORDER member function that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the ORDER method is null, then the ORDER method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the ORDER method map_order_function_spec is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

A subtype can neither define nor override an ORDER method.

**varray_type_def**
The `varray_type_def` clause lets you create the type as an ordered set of elements, each of which has the same data type.

**Restrictions on Varray Types** You can create a VARRAY type of XMLType or of a LOB type for procedural purposes, for example, in PL/SQL or in view queries. However, database storage of such a varray is not supported, so you cannot create an object table or an object type column of such a varray type.

See Also: Varray Type Example on page 14-80

**nested_table_type_def**
The `nested_table_type_def` clause lets you create a named nested table of type datatype.

See Also:
- Nested Table Type Example on page 14-80
- Nested Table Type Containing a Varray on page 14-80

**Examples**

**Object Type Examples** The following example shows how the sample type `customer_typ` was created for the sample Order Entry (oe) schema. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE customer_typ_demo AS OBJECT
( customer_id        NUMBER(6)
, cust_first_name    VARCHAR2(20)
, cust_last_name     VARCHAR2(20)
, cust_address       CUST_ADDRESS_TYP
, phone_numbers      PHONE_LIST_TYP
, nls_language       VARCHAR2(3)
, nls_territory      VARCHAR2(30)
, credit_limit       NUMBER(9,2)
, cust_email         VARCHAR2(30)
, cust_orders        ORDER_LIST_TYP
) ;
```
In the following example, the `data_typ1` object type is created with one member function `prod`, which is implemented in the `CREATE TYPE BODY` statement:

```sql
CREATE TYPE data_typ1 AS OBJECT
  ( year NUMBER,
    MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
  );
/
CREATE TYPE BODY data_typ1 IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
    BEGIN
      RETURN (year + invent);
    END;
  END;
/
```

**Subtype Example** The following statement shows how the subtype `corporate_customer_typ` in the sample `oe` schema was created. It is based on the `customer_typ` supertype created in the preceding example and adds the `account_mgr_id` attribute. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```sql
CREATE TYPE corporate_customer_typ_demo UNDER customer_typ
  ( account_mgr_id     NUMBER(6)
  );
```

**SQLJ Object Type Example** The following examples create a SQLJ object type and subtype. The `address_t` type maps to the Java class `Examples.Address`. The subtype `long_address_t` maps to the Java class `Examples.LongAddress`. The examples specify SQLData as the mechanism used to create the Java instance of these types. Each of the functions in these type specifications has a corresponding implementation in the Java class.

```sql
CREATE TYPE address_t AS OBJECT
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city VARCHAR, state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME 'create (java.lang.String, java.lang.String, java.lang.String, int) return Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR, state VARCHAR, zip NUMBER) RETURN address_t
  )
```

**See Also:** Oracle Database Object-Relational Developer’s Guide for the Java implementation of the functions in these type specifications.
CREATE TYPE Statement

SQL Statements for Stored PL/SQL Units 14-73

EXTERNAL NAME
'create (java.lang.String, java.lang.String, java.lang.String, int) return
Examples.Address',
MEMBER FUNCTION to_string RETURN VARCHAR
   EXTERNAL NAME 'tojava.lang.String() return java.lang.String',
MEMBER FUNCTION strip RETURN SELF AS RESULT
   EXTERNAL NAME 'removeLeadingBlanks () return Examples.Address'
} NOT FINAL;
/

CREATE OR REPLACE TYPE long_address_t
UNDER address_t
EXTERNAL NAME 'Examples.LongAddress' LANGUAGE JAVA
USING SQLData{
   street2_attr VARCHAR(250) EXTERNAL NAME 'street2',
   country_attr VARCHAR (200) EXTERNAL NAME 'country',
   address_code_attr VARCHAR (50) EXTERNAL NAME 'addrCode',
   STATIC FUNCTION create_address RETURN long_address_t
      EXTERNAL NAME 'create() return Examples.LongAddress',
   STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
      state VARCHAR, country VARCHAR, addrs_cd VARCHAR)
      RETURN long_address_t
      EXTERNAL NAME 'create(java.lang.String, java.lang.String,
      java.lang.String, java.lang.String, java.lang.String)
      return Examples.LongAddress',
   STATIC FUNCTION construct RETURN long_address_t
      EXTERNAL NAME 'Examples.LongAddress() return Examples.LongAddress',
   STATIC FUNCTION create_longaddress {
      street VARCHAR, city VARCHAR, state VARCHAR, country VARCHAR,
      addrs_cd VARCHAR} return long_address_t
   EXTERNAL NAME
      'Examples.LongAddress (java.lang.String, java.lang.String,
      java.lang.String, java.lang.String, java.lang.String)
      return Examples.LongAddress',
   MEMBER FUNCTION get_country RETURN VARCHAR
      EXTERNAL NAME 'country_with_code () return java.lang.String'
};
/

Type Hierarchy Example  The following statements create a type hierarchy. Type
employee_t inherits the name and ssn attributes from type person_t and in
addition has department_id and salary attributes. Type part_time_emp_t
inherits all of the attributes from employee_t and, through employee_t, those of
person_t and in addition has a num_hrs attribute. Type part_time_emp_t is final
by default, so no further subtypes can be created under it.

CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;
/

CREATE TYPE employee_t UNDER person_t
   (department_id NUMBER, salary NUMBER) NOT FINAL;
/

CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
You can use type hierarchies to create substitutable tables and tables with substitutable columns.

**Varray Type Example**  The following statement shows how the `phone_list_typ` varray type with five elements in the sample oe schema was created. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE phone_list_typ_demo AS VARRAY(5) OF VARCHAR2(25);
```

**Nested Table Type Example**  The following example from the sample schema pm creates the table type `textdoc_tab` of object type `textdoc_typ`:

```
CREATE TYPE textdoc_typ AS OBJECT
   ( document_typ   VARCHAR2(32)
   , formatted_doc  BLOB
   ) ;
CREATE TYPE textdoc_tab AS TABLE OF textdoc_typ;
```

**Nested Table Type Containing a Varray**  The following example of multilevel collections is a variation of the sample table oe.customers. In this example, the `cust_address` object column becomes a nested table column with the `phone_list_typ` varray column embedded in it. The `phone_list_typ` type was created in Varray Type Example on page 14-80.

```
CREATE TYPE cust_address_typ2 AS OBJECT
   ( street_address     VARCHAR2(40)
   , postal_code        VARCHAR2(10)
   , city               VARCHAR2(30)
   , state_province     VARCHAR2(10)
   , country_id         CHAR(2)
   , phone              phone_list_typ_demo
   ) ;
CREATE TYPE cust_nt_address_typ
   AS TABLE OF cust_address_typ2;
```

**Constructor Example**  This example invokes the system-defined constructor to construct the `demo_typ` object and insert it into the `demo_tab` table:

```
CREATE TYPE demo_typ1 AS OBJECT (a1 NUMBER, a2 NUMBER);
CREATE TABLE demo_tab1 (b1 NUMBER, b2 demo_typ1);
INSERT INTO demo_tab1 VALUES (1, demo_typ1(2,3));
```

**Creating a Member Method: Example**  The following example invokes method `constructor col.get_square`. First the type is created:

```
CREATE TYPE demo_typ2 AS OBJECT (a1 NUMBER,
   MEMBER FUNCTION get_square RETURN NUMBER);
CREATE TABLE demo_tab2 (col demo_typ2);
INSERT INTO demo_tab2 VALUES (demo_typ2(2));
```

*See Also:* Oracle Database Object-Relational Developer’s Guide for more information about constructors.
The type body is created to define the member function, and the member method is invoked:

```sql
CREATE TYPE BODY demo_typ2 IS
    MEMBER FUNCTION get_square
        RETURN NUMBER
    IS
        x NUMBER;
        BEGIN
            SELECT c.col.a1*c.col.a1 INTO x
                FROM demo_tab2 c;
            RETURN (x);
        END;
    END;
END;
```

```sql
SELECT t.col.get_square() FROM demo_tab2 t;
```

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

**Creating a Static Method: Example**  The following example changes the definition of the `employee_t` type to associate it with the `construct_emp` function. The example first creates an object type `department_t` and then an object type `employee_t` containing an attribute of type `department_t`:

```sql
CREATE OR REPLACE TYPE department_t AS OBJECT {
    deptno number(10),
    dname CHAR(30));

CREATE OR REPLACE TYPE employee_t AS OBJECT {
    empid RAW(16),
    ename CHAR(31),
    dept REF department_t,
    STATIC function construct_emp
        (name VARCHAR2, dept REF department_t)
        RETURN employee_t
    ) ;

This statement requires the following type body statement.

```sql
CREATE OR REPLACE TYPE BODY employee_t IS
    STATIC FUNCTION construct_emp
        (name VARCHAR2, dept REF department_t)
        RETURN employee_t
    IS
        BEGIN
            return employee_t(SYS_GUID(),name,dept);
        END;
    END;
END;
```

Next create an object table and insert into the table:

```sql
CREATE TABLE emptab OF employee_t;
INSERT INTO emptab
    VALUES (employee_t.construct_emp('John Smith', NULL));
```
Related Topics

- ALTER TYPE Statement on page 14-15
- Collection on page 13-22
- CREATE TYPE BODY Statement on page 14-83
- DROP TYPE Statement on page 14-95
- Defining Collection Types on page 5-7

See Also: Oracle Database Object-Relational Developer’s Guide for more information about objects, incomplete types, varrays, and nested tables
CREATE TYPE BODY Statement

The CREATE TYPE BODY defines or implements the member methods defined in the object type specification. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE Statement on page 14-65 specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods that implement the type.

For each method specified in an object type specification for which you did not specify the call_spec, you must specify a corresponding method body in the object type body.

---
Note: If you create a SQLJ object type, then specify it as a Java class.
---

Prerequisites

Every member declaration in the CREATE TYPE specification for object types must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create an object type in another user’s schema, you must have the CREATE ANY TYPE system privilege. To replace an object type in another user’s schema, you must have the DROP ANY TYPE system privilege.

Syntax

create_type_body ::=  

CREATE | OR REPLACE TYPE BODY (schema) type_name IS AS subprog_decl_in_type map_order_func_declaration, END;  

(subprog_decl_in_type ::= on page 14-84, map_order_func_declaration ::= on page 14-85)

subprog_decl_in_type ::=  

MEMBER | STATIC proc_decl_in_type func_decl_in_type constructor_declaration

proc_decl_in_type ::=  

PROCEDURE name (parameter datatype) IS AS declare_section body call_spec
CREATE TYPE BODY Statement

(declare_section ::= on page 13-8, body ::= on page 13-11, call_spec ::= on page 14-85)

func_decl_in_type ::= 

```
FUNCTION name (parameter datatype RETURN datatype)

IS AS
decall_section
body

call_spec
```

(declare_section ::= on page 13-8, body ::= on page 13-11, call_spec ::= on page 14-85)

constructor_declaration ::= 

```
FINAL INSTANTIABLE CONSTRUCTOR FUNCTION datatype

SELF IN OUT datatype

parameter datatype

RETURN SELF AS RESULT

IS AS
decall_section
body

call_spec
```

map_order_func_declaration ::= 

```
MAP ORDER MEMBER function_declaration
```

call_spec ::= 

```
LANGUAGE Java_declaration C_declaration
```

Java_declaration ::= 

```
JAVA NAME string
```
OR REPLACE
Specify OR REPLACE to re-create the type body if it already exists. Use this clause to change the definition of an existing type body without first dropping it.

Users previously granted privileges on the re-created object type body can use and reference the object type body without being granted privileges again.

You can use this clause to add new member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE statement.

schema
Specify the schema to contain the type body. If you omit schema, then the database creates the type body in your current schema.

type_name
Specify the name of an object type.

subprog_decl_in_type
Specify the type of function or procedure subprogram associated with the object type specification.

You must define a corresponding method name and optional parameter list in the object type specification for each procedure or function declaration. For functions, you also must specify a return type.

proc_decl_in_type, func_decl_in_type  Declare a procedure or function subprogram.

classifier
Declare a user-defined constructor subprogram. The RETURN clause of a constructor function must be RETURN SELF AS RESULT. This setting indicates that the most specific type of the value returned by the constructor function is the same as the most specific type of the SELF argument that was passed in to the constructor function.

See Also:

- CREATE TYPE Statement on page 14-65 for a list of restrictions on user-defined functions
- Overloading PL/SQL Subprogram Names on page 8-12 for information about overloading subprogram names
- Oracle Database Object-Relational Developer’s Guide for information about and examples of user-defined constructors
**DECLARE_SECTION**
Declares items that are local to the procedure or function.

**BODY**
Procedure or function statements.

**CALL_SPEC** Specify the call specification that maps a Java or C method name, parameter types, and return type to their SQL counterparts.

The *Java_declaration* string identifies the Java implementation of the method.

**See Also:**
- *Oracle Database Java Developer’s Guide* for an explanation of the parameters and semantics of the *Java_declaration*
- *Oracle Database Advanced Application Developer’s Guide* for information about calling external procedures

**MAP_ORDER_FUNC_DECLARATION**
You can declare either one MAP method or one ORDER method, regardless of how many MEMBER or STATIC methods you declare. If you declare either a MAP or ORDER method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

**MAP MEMBER Clause**
Specify MAP MEMBER to declare or implement a MAP member function that returns the relative position of a given instance in the ordering of all instances of the object. A MAP method is called implicitly and specifies an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the MAP method is null, then the MAP method returns null and the method is not invoked.

An object type body can contain only one MAP method, which must be a function. The MAP function can have no arguments other than the implicit *SELF* argument.

**ORDER MEMBER Clause**
Specify ORDER MEMBER to specify an ORDER member function that takes an instance of an object as an explicit argument and the implicit *SELF* argument and returns either a negative integer, zero, or a positive integer, indicating that the implicit *SELF* argument is less than, equal to, or greater than the explicit argument, respectively.

If either argument to the ORDER method is null, then the ORDER method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the database invokes the ORDER MEMBER *func_decl_in_type*.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.
func_decl_in_type  Declare a function subprogram. See CREATE PROCEDURE Statement on page 14-46 and CREATE FUNCTION Statement on page 14-29 for the full syntax with all possible clauses.

AS EXTERNAL  AS EXTERNAL is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle recommends that you use the call_spec syntax with the C_declaration.

Examples

Several examples of creating type bodies appear in the Examples section of CREATE TYPE Statement on page 14-65. For an example of re-creating a type body, see Adding a Member Function: Example on page 14-27.

Related Topics

- CREATE TYPE Statement on page 14-65
- DROP TYPE BODY Statement on page 14-97
- CREATE FUNCTION Statement on page 14-29
- CREATE PROCEDURE Statement on page 14-46
DROP FUNCTION Statement

The **DROP FUNCTION** statement drops a standalone stored function from the database.

---

**Note:** Do not use this statement to drop a function that is part of a package. Instead, either drop the entire package using the **DROP PACKAGE Statement** on page 14-91 or redefine the package without the function using the **CREATE PACKAGE Statement** on page 14-39 with the **OR REPLACE** clause.

---

**Prerequisites**

The function must be in your own schema or you must have the **DROP ANY PROCEDURE** system privilege.

**Syntax**

```
drop_function ::= [ DROP FUNCTION ] schema function_name ;
```

**Keyword and Parameter Descriptions**

- **schema**
  
  Specify the schema containing the function. If you omit `schema`, then the database assumes the function is in your own schema.

- **function_name**
  
  Specify the name of the function to be dropped.

  The database invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped function.

  If any statistics types are associated with the function, then the database disassociates the statistics types with the **FORCE** option and drops any user-defined statistics collected with the statistics type.

**See Also:**

- Oracle Database SQL Language Reference for information about the **ASSOCIATE STATISTICS** statement
- Oracle Database SQL Language Reference for information about the **DISASSOCIATE STATISTICS** statement

**Example**

**Dropping a Function: Example**  

The following statement drops the function `SecondMax` in the sample schema `oe` and invalidates all objects that depend upon `SecondMax`:
DROP FUNCTION oe.SecondMax;

See Also: Creating Aggregate Functions: Example on page 14-37 for information about creating the SecondMax function

Related Topics
- ALTER FUNCTION Statement on page 14-3
- CREATE FUNCTION Statement on page 14-29
The `DROP PACKAGE` statement drops a stored package from the database. This statement drops the body and specification of a package.

**Prerequisites**

The package must be in your own schema or you must have the `DROP ANY PROCEDURE` system privilege.

**Syntax**

```
drop_package::=
```

**Keyword and Parameter Descriptions**

**BODY**

Specify `BODY` to drop only the body of the package. If you omit this clause, then the database drops both the body and specification of the package.

When you drop only the body of a package but not its specification, the database does not invalidate dependent objects. However, you cannot call one of the procedures or stored functions declared in the package specification until you re-create the package body.

**schema**

Specify the schema containing the package. If you omit `schema`, then the database assumes the package is in your own schema.

**package**

Specify the name of the package to be dropped.

The database invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, then the database disassociates the statistics types with the `FORCE` clause and drops any user-defined statistics collected with the statistics types.
DROP PACKAGE Statement

Example

Dropping a Package: Example The following statement drops the specification and body of the emp_mgmt package, which was created in Creating a Package Body: Example on page 14-43, invalidating all objects that depend on the specification:

DROP PACKAGE emp_mgmt;

Related Topics

- ALTER PACKAGE Statement on page 14-6
- CREATE PACKAGE Statement on page 14-39
- CREATE PACKAGE BODY Statement on page 14-42

See Also:

- Oracle Database SQL Language Reference for information about the ASSOCIATE STATISTICS statement
- Oracle Database SQL Language Reference for information about the DISASSOCIATE STATISTICS statement
DROP PROCEDURE Statement

The DROP PROCEDURE statement drops a standalone stored procedure from the database.

**Note:** Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the DROP PACKAGE Statement on page 14-91, or redefine the package without the procedure using the CREATE PACKAGE Statement on page 14-39 with the OR REPLACE clause.

**Prerequisites**

The procedure must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

**Syntax**

\[\text{drop\_procedure::=} \quad \text{DROP}\ \text{PROCEDURE} \quad \text{schema}\ \text{procedure}\ \quad \text{;}\]

**Keyword and Parameter Descriptions**

- **schema**
  
  Specify the schema containing the procedure. If you omit `schema`, then the database assumes the procedure is in your own schema.

- **procedure**
  
  Specify the name of the procedure to be dropped.
  
  When you drop a procedure, the database invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

**Example**

**Dropping a Procedure: Example** The following statement drops the procedure `remove_emp` owned by the user `hr` and invalidates all objects that depend upon `remove_emp`:

```
DROP PROCEDURE hr.remove_emp;
```

**Related Topics**

- [ALTER PROCEDURE Statement](#) on page 14-9
- [CREATE PROCEDURE Statement](#) on page 14-46
**DROP TRIGGER Statement**

The **DROP TRIGGER** statement drops a database trigger from the database.

**Prerequisites**

The trigger must be in your own schema or you must have the **DROP ANY TRIGGER** system privilege. To drop a trigger on **DATABASE** in another user’s schema, you must also have the **ADMINISTER DATABASE TRIGGER** system privilege.

**Syntax**

```sql
drop_trigger ::= 
```

- `DROP TRIGGER` schema trigger ;

**Keyword and Parameter Descriptions**

- **schema**
  Specify the schema containing the trigger. If you omit `schema`, then the database assumes the trigger is in your own schema.

- **trigger**
  Specify the name of the trigger to be dropped. the database removes it from the database and does not fire it again.

**Example**

**Dropping a Trigger: Example**

The following statement drops the `salary_check` trigger in the schema `hr`:

```sql
DROP TRIGGER hr.salary_check;
```

**Related Topics**

- [ALTER TRIGGER Statement](#) on page 14-12
- [CREATE TRIGGER Statement](#) on page 14-51
The DROP TYPE statement drops the specification and body of an object type, a varray, or a nested table type.

**Prerequisites**

The object type, varray, or nested table type must be in your own schema or you must have the DROP ANY TYPE system privilege.

**Syntax**

\[
drop\_type::=\]

![Syntax Diagram](image)

**Keyword and Parameter Descriptions**

**schema**

Specify the schema containing the type. If you omit `schema`, then the database assumes the type is in your own schema.

**type_name**

Specify the name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If `type_name` is a supertype, then this statement will fail unless you also specify FORCE. If you specify FORCE, then the database invalidates all subtypes depending on this supertype.

If `type_name` is a statistics type, then this statement will fail unless you also specify FORCE. If you specify FORCE, then the database first disassociates all objects that are associated with `type_name` and then drops `type_name`.

**See Also:**

- *Oracle Database SQL Language Reference* for information about the ASSOCIATE STATISTICS statement
- *Oracle Database SQL Language Reference* for information about the DISASSOCIATE STATISTICS statement

If `type_name` is an object type that has been associated with a statistics type, then the database first attempts to disassociate `type_name` from the statistics type and then drops `type_name`. However, if statistics have been collected using the statistics type, then the database will be unable to disassociate `type_name` from the statistics type, and this statement will fail.

If `type_name` is an implementation type for an indextype, then the indextype will be marked INVALID.
If `type_name` has a public synonym defined on it, then the database will also drop the synonym.

Unless you specify `FORCE`, you can drop only object types, nested tables, or varray types that are standalone schema objects with no dependencies. This is the default behavior.

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

**FORCE**

Specify `FORCE` to drop the type even if it has dependent database objects. The database marks `UNUSED` all columns dependent on the type to be dropped, and those columns become inaccessible.

**Caution:** Oracle does not recommend that you specify `FORCE` to drop object types with dependencies. This operation is not recoverable and could cause the data in the dependent tables or columns to become inaccessible.

**VALIDATE**

If you specify `VALIDATE` when dropping a type, then the database checks for stored instances of this type within substitutable columns of any of its supertypes. If no such instances are found, then the database completes the drop operation.

This clause is meaningful only for subtypes. Oracle recommends the use of this option to safely drop subtypes that do not have any explicit type or table dependencies.

**Example**

**Dropping an Object Type: Example**  
The following statement removes object type `person_t`. See Type Hierarchy Example on page 14-80 for the example that creates this object type. Any columns that are dependent on `person_t` are marked `UNUSED` and become inaccessible.

```
DROP TYPE person_t FORCE;
```
DROP TYPE BODY Statement

The DROP TYPE BODY statement drops the body of an object type, varray, or nested table type. When you drop a type body, the object type specification still exists, and you can re-create the type body. Prior to re-creating the type body, you can still use the object type, although you cannot call the member functions.

Prerequisites

The object type body must be in your own schema or you must have the DROP ANY TYPE system privilege.

Syntax

drop_type_body ::= 

Keyword and Parameter Descriptions

schema
Specify the schema containing the object type. If you omit schema, then the database assumes the object type is in your own schema.

type_name
Specify the name of the object type body to be dropped.

Restriction on Dropping Type Bodies

You can drop a type body only if it has no type or table dependencies.

Example

Dropping an Object Type Body: Example

The following statement removes object type body data_typ1. See Object Type Examples on page 14-78 for the example that creates this object type.

DROP TYPE BODY data_typ1;

Related Topics

- ALTER TYPE Statement on page 14-15
- CREATE TYPE Statement on page 14-65
- CREATE TYPE BODY Statement on page 14-83
This appendix explains what wrapping is, why you wrap PL/SQL code, and how to do it.

Topics:
- Overview of Wrapping
- Guidelines for Wrapping
- Limitations of Wrapping
- Wrapping PL/SQL Code with wrap Utility
- Wrapping PL/QL Code with DBMS_DDL Subprograms

Overview of Wrapping

Wrapping is the process of hiding PL/SQL source code. Wrapping helps to protect your source code from business competitors and others who might misuse it.

You can wrap PL/SQL source code with either the wrap utility or DBMS_DDL subprograms. The wrap utility wraps a single source file, such as a SQL*Plus script. The DBMS_DDL subprograms wrap a single dynamically generated PL/SQL unit, such as a single CREATE PROCEDURE statement.

Wrapped source files can be moved, backed up, and processed by SQL*Plus and the Import and Export utilities, but they are not visible through the static data dictionary views *_SOURCE.

Note: Wrapping a file that is already wrapped has no effect on the file.

Guidelines for Wrapping

- Wrap only the body of a package or object type, not the specification.
  This allows other developers to see the information they must use the package or type, but prevents them from seeing its implementation.

- Wrap code only after you have finished editing it.
  You cannot edit PL/SQL source code inside wrapped files. Either wrap your code after it is ready to ship to users or include the wrapping operation as part of your build environment.
To change wrapped PL/SQL code, edit the original source file and then wrap it again.

- Before distributing a wrapped file, view it in a text editor to be sure that all important parts are wrapped.

**Limitations of Wrapping**

- Wrapping is not a secure method for hiding passwords or table names.
  Wrapping a PL/SQL unit prevents most users from examining the source code, but might not stop all of them.
- Wrapping does not hide the source code for triggers.
  To hide the workings of a trigger, write a one-line trigger that invokes a wrapped subprogram.
- Wrapping does not detect syntax or semantic errors.
  Wrapping detects only tokenization errors (for example, runaway strings), not syntax or semantic errors (for example, nonexistent tables or views). Syntax or semantic errors are detected during PL/SQL compilation or when executing the output file in SQL*Plus.
- Wrapped PL/SQL units are not downward-compatible.
  Wrapped PL/SQL units are upward-compatible between Oracle Database releases, but are not downward-compatible. For example, you can load files processed by the V8.1.5 `wrap` utility into a V8.1.6 Oracle Database, but you cannot load files processed by the V8.1.6 `wrap` utility into a V8.1.5 Oracle Database.

**See Also:**

- Limitations of the `wrap` Utility on page A-4
- Limitation of the DBMS_DDL.WRAP Function on page A-6

**Wrapping PL/SQL Code with `wrap` Utility**

The `wrap` utility processes an input SQL file and wraps only the PL/SQL units in the file, such as a package specification, package body, function, procedure, type specification, or type body. It does not wrap PL/SQL content in anonymous blocks or triggers or non-PL/SQL code.

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
```
**Wrapping PL/SQL Code with wrap Utility**

output_file is the name of the wrapped 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:

```sql
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:

```sql
wrap iname=/mydir/myfile oname=/yourdir/yourfile.out
```

**Note:** If input_file is already wrapped, output_file will be identical to input_file.

Topics:

- Input and Output Files for the PL/SQL wrap Utility
- Running the wrap Utility
- Limitations of the wrap Utility

**Input and Output Files for the PL/SQL wrap Utility**

The input file can contain any combination of SQL statements. Most statements are passed through unchanged. `CREATE` statements that define subprograms, packages, or object types are wrapped; their bodies are replaced by a scrambled form that the PL/SQL compiler understands.

The following `CREATE` statements are wrapped:

- `CREATE [OR REPLACE] FUNCTION function_name`
- `CREATE [OR REPLACE] PROCEDURE procedure_name`
- `CREATE [OR REPLACE] PACKAGE package_name`
- `CREATE [OR REPLACE] PACKAGE BODY package_name`
- `CREATE [OR REPLACE] TYPE type_name AS OBJECT`
- `CREATE [OR REPLACE] TYPE type_name UNDER type_name`
- `CREATE [OR REPLACE] TYPE BODY type_name`

The `CREATE [OR REPLACE] TRIGGER` statement, and `[DECLARE] BEGIN-END` anonymous blocks, are not wrapped. All other SQL statements are passed unchanged to the output file.

All comment lines in the unit being wrapped are deleted, except for those in a `CREATE OR REPLACE` header and C-style comments (delimited by /* */).

The output file is a text file, which you can run as a script in SQL*Plus to set up your PL/SQL subprograms and packages. Run a wrapped file as follows:

```sql
SQL> @wrapped_file_name.plb;
```

**Running the wrap Utility**

For example, assume that the `wrap_test.sql` file contains the following:

```sql
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;
```

Note: If input_file is already wrapped, output_file will be identical to input_file.
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();

Limitations of the wrap Utility

- The PL/SQL code to be wrapped cannot include substitution variables using the
  SQL*Plus DEFINE notation.

  Wrapped source code is parsed by the PL/SQL compiler, not by SQL*Plus.

- The wrap utility removes most comments from wrapped files.

  See Input and Output Files for the PL/SQL wrap Utility on page A-3.

Wrapping PL/QL Code with DBMS_DDL Subprograms

The DBMS_DDL package contains procedures for wrapping a single PL/SQL unit, such
as a package specification, package body, function, procedure, type specification, or
Type body. These overloaded subprograms provide a mechanism for wrapping
dynamically generated PL/SQL units that are created in a database.

The DBMS_DDL package contains the WRAP functions and the CREATE_WRAPPED
procedures. The CREATE_WRAPPED both wraps the text and creates the PL/SQL unit.
When invoking the wrap procedures, use the fully qualified package name,
SYS.DBMS_DDL, to avoid any naming conflicts and the possibility that someone might
create a local package called DBMS_DDL or define the DBMS_DDL public synonym. The
input CREATE OR REPLACE statement 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.

Note: Wrapping a PL/SQL unit that is already wrapped has no effect
on the unit.
Wrapping PL/QL Code with DBMS_DDL Subprograms

Topics:

- Using DBMS_DDL.CREATE_WRAPPED Procedure
- Limitation of the DBMS_DDL.WRAP Function

See Also: Oracle Database PL/SQL Packages and Types Reference for information about the DBMS_DDL package

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 DBMS_DDL.CREATE_WRAPPED Procedure to Wrap a Package

DECLARE
    package_text VARCHAR2(32767); -- text for creating package spec & body

FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS
    BEGIN
        RETURN 'CREATE PACKAGE ' || pkgname || ' AS
        PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
        PROCEDURE fire_employee (emp_id NUMBER);
        END ' || pkgname || ';';
    END generate_spec;

FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2 AS
    BEGIN
        RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
        PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
            BEGIN
                UPDATE employees
                SET salary = salary + amount WHERE employee_id = emp_id;
                END raise_salary;
                PROCEDURE fire_employee (emp_id NUMBER) IS
                    BEGIN
                        DELETE FROM employees WHERE employee_id = emp_id;
                        END fire_employee;
            END ' || pkgname || ';';
    END generate_body;

BEGIN
    -- Generate package spec
    package_text := generate_spec('emp_actions')

    -- Create wrapped package spec
    DBMS_DDL.CREATE_WRAPPED(package_text);

    -- Generate package body
    package_text := generate_body('emp_actions')

    -- Create wrapped package body
    DBMS_DDL.CREATE_WRAPPED(package_text);
END;
/

-- Invoke procedure from wrapped package
CALL emp_actions.raise_salary(120, 100);
When you check the static data dictionary views *_SOURCE, the source is wrapped, or hidden, so that others cannot view the code details. For example:

```sql
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

The resulting output is similar to the following:

```
TEXT
--------------------------------------------------------------------
PACKAGE emp_actions WRAPPED
a000000
lf
abcd
...
```

**Limitation of the DBMS_DDL.WRAP Function**

If you invoke DBMS_SQLPARSE (when using an overload where the statement formal has data type VARCHAR2A or VARCHAR2S for text which exceeds 32767 bytes) on the output of DBMS_DDL.WRAP, then you must set the LFFLG parameter to FALSE. Otherwise DBMS_SQLPARSE adds newlines to the wrapped unit which corrupts the unit.
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 data type Client because PL/SQL names are not case sensitive.

Example B–1  Resolving Global and Local Variable Names

```
BEGIN
  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;
```
You can refer to data type `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)
      END f1;
END pkg1;
/```
Examples of Qualified Names and Dot Notation

n := pkg1.v1.a;
   -- (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;
/

An outside reference to a private variable declared in a function body is not legal. For example, an outside reference to the variable n declared in function f1, such as hr.pkg1.f1.n from function f2, raises an exception. See Private and Public Items in 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. Some of the combinations are:

- Field or attribute of a function return value, for example:
  func_name().field_name
  func_name().attribute_name

- Schema object owned by another schema, for example:
  schema_name.table_name
  schema_name.procedure_name()
  schema_name.type_name.member_name()

- Package object owned by another user, for example:
  schema_name.package_name.procedure_name()
  schema_name.package_name.record_name.field_name

- Record containing 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 about SQL naming rules, see *Oracle Database SQL Language Reference*.

What is Capture?

When a declaration or type definition in another scope prevents the compiler from resolving a reference correctly, that declaration or definition is said to capture the reference. Capture is usually the result of migration or schema evolution. There are three kinds of capture: inner, same-scope, and outer. Inner and same-scope capture apply only in SQL scope.

Topics:

- Inner Capture
- Same-Scope Capture
- Outer Capture

Inner Capture

An inner capture occurs when a name in an inner scope no longer refers to an entity in an outer scope:

- The name might now resolve to an entity in an inner scope.
- The program might cause an error, if some part of the identifier is captured in an inner scope and the complete reference cannot be resolved.

If the reference points to a different but valid name, you might not know why the program is acting differently.

In the following example, the reference to `col2` in the inner `SELECT` statement binds to column `col2` in table `tab1` because table `tab2` has no column named `col2`:

```sql
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`:

```sql
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:

```sql
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:

```sql
SELECT * FROM hr.tab1 p1
  WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);
```

**Same-Scope Capture**

In SQL scope, a same-scope capture occurs when a column is added to one of two tables used in a join, so that the same column name exists in both tables. Previously, you could refer to that column name in a join query. To avoid an error, now you must qualify the column name with the table name.

**Outer Capture**

An outer capture occurs when a name in an inner scope, which once resolved to an entity in an inner scope, is resolved to an entity in an outer scope. SQL and PL/SQL are designed to prevent outer captures. You need not take any action to avoid this condition.

**Avoiding Inner Capture in DML Statements**

You can avoid inner capture in DML statements by following these rules:
Avoiding Inner Capture in DML Statements

- 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 tbl (col1 t1);

BEGIN
  -- following inserts are allowed without an alias
  INSERT INTO tbl VALUES ( t1(10) );
  INSERT INTO tbl VALUES ( t1(20) );
  INSERT INTO tbl VALUES ( t1(30) );
END;
/
BEGIN
  UPDATE tbl SET col1.x = 10
  WHERE col1.x = 20; -- error, not allowed
END;
/
BEGIN
  UPDATE tbl SET tbl.col1.x = 10
  WHERE tbl.col1.x = 20; -- not allowed
END;
/
BEGIN
  UPDATE hr.tbl SET hr.tbl.col1.x = 10
  WHERE hr.tbl.col1.x = 20; -- not allowed
END;
/
BEGIN -- following allowed with table alias
  UPDATE hr.tbl 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 tb1 t
  WHERE t.col1.x = 30;
END;
/

BEGIN
  DELETE FROM tb1
  WHERE tb1.col1.x = 10; -- not allowed
END;
/

BEGIN -- following allowed with table alias
  DELETE FROM tb1 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. For example:

```sql
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 -- following allowed with table alias
    DELETE FROM ot1 o
    WHERE VALUE(o) = (t1(20));
END;
/

Avoiding Inner Capture in DML Statements
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 system code. Both the DIANA and system code for a subprogram or package are stored in the database. At run time, they are loaded into the shared memory pool. The DIANA is used to compile dependent subprograms; the system code 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>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind variables passed to a program unit</td>
<td>32768</td>
</tr>
<tr>
<td>exception handlers in a program unit</td>
<td>65536</td>
</tr>
<tr>
<td>fields in a record</td>
<td>65536</td>
</tr>
<tr>
<td>levels of block nesting</td>
<td>255</td>
</tr>
<tr>
<td>levels of record nesting</td>
<td>32</td>
</tr>
<tr>
<td>levels of subquery nesting</td>
<td>254</td>
</tr>
<tr>
<td>levels of label nesting</td>
<td>98</td>
</tr>
<tr>
<td>levels of nested collections</td>
<td>no predefined limit</td>
</tr>
<tr>
<td>magnitude of a PLS_INTEGER or BINARY_</td>
<td>-2147483648..2147483647</td>
</tr>
<tr>
<td>INTEGERvalue</td>
<td></td>
</tr>
<tr>
<td>number of formal parameters in an explicit</td>
<td>65536</td>
</tr>
<tr>
<td>cursor, function, or procedure</td>
<td></td>
</tr>
<tr>
<td>objects referenced by a program unit</td>
<td>65536</td>
</tr>
<tr>
<td>precision of a FLOAT value (binary digits)</td>
<td>126</td>
</tr>
<tr>
<td>precision of a NUMBER value (decimal digits)</td>
<td>38</td>
</tr>
<tr>
<td>precision of a REAL value (binary digits)</td>
<td>63</td>
</tr>
</tbody>
</table>
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';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>SOURCE_SIZE</th>
<th>PARSED_SIZE</th>
<th>CODE_SIZE</th>
<th>ERROR_SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKG1</td>
<td>PACKAGE</td>
<td>46</td>
<td>165</td>
<td>119</td>
<td>0</td>
</tr>
<tr>
<td>PKG1</td>
<td>PACKAGE BODY</td>
<td>82</td>
<td>0</td>
<td>139</td>
<td>0</td>
</tr>
</tbody>
</table>

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 more information about the limits on data types, see Chapter 3, "PL/SQL Data Types." For limits on collection subscripts, see Referencing Collection Elements on page 5-12.
Both **reserved words** and **keywords** have special meaning in PL/SQL. The difference between reserved words and keywords is that you cannot use reserved words as identifiers. You can use keywords 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**

<table>
<thead>
<tr>
<th>Begins with</th>
<th>Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ALL, ALTER, AND, ANY, AS, ASC, AT</td>
</tr>
<tr>
<td>B</td>
<td>BEGIN, BETWEEN, BY</td>
</tr>
<tr>
<td>C</td>
<td>CASE, CHECK, CLUSTER, CLUSTERS, COLAUTH, COLUMNS, COMPRESS, CONNECT, CRASH, CREATE, CURRENT</td>
</tr>
<tr>
<td>D</td>
<td>DECLARE, DEFAULT, DELETE, DESC, DISTINCT, DROP</td>
</tr>
<tr>
<td>E</td>
<td>ELSE, END, EXCEPTION, EXCLUSIVE, EXISTS</td>
</tr>
<tr>
<td>F</td>
<td>FETCH, FOR, FROM</td>
</tr>
<tr>
<td>G</td>
<td>GOTO, GRANT, GROUP</td>
</tr>
<tr>
<td>H</td>
<td>HAVING</td>
</tr>
<tr>
<td>I</td>
<td>IDENTIFIED, IF, IN, INDEX, INDEXES, INSERT, INTERSECT, INTO, IS</td>
</tr>
<tr>
<td>L</td>
<td>LIKE, LOCK</td>
</tr>
<tr>
<td>M</td>
<td>MINUS, MODE</td>
</tr>
<tr>
<td>N</td>
<td>NOCOMPRESS, NOT, NOWAIT, NULL</td>
</tr>
<tr>
<td>O</td>
<td>OF, ON, OPTION, OR, ORDER, OVERLAPS</td>
</tr>
<tr>
<td>P</td>
<td>PRIOR, PROCEDURE, PUBLIC</td>
</tr>
<tr>
<td>R</td>
<td>RESOURCE, REVOKE</td>
</tr>
<tr>
<td>S</td>
<td>SELECT, SHARE, SIZE, SQL, START</td>
</tr>
<tr>
<td>T</td>
<td>TABAUTH, TABLE, THEN, TO</td>
</tr>
<tr>
<td>U</td>
<td>UNION, UNIQUE, UPDATE</td>
</tr>
<tr>
<td>V</td>
<td>VALUES, VIEW, VIEWS</td>
</tr>
<tr>
<td>W</td>
<td>WHEN, WHERE, WITH</td>
</tr>
<tr>
<td>Begins with</td>
<td>Keywords</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>A</td>
<td>A, ADD, AGENT, AGGREGATE, ARRAY, ATTRIBUTE, AUTHID, AVG</td>
</tr>
<tr>
<td>B</td>
<td>BFILE_BASE, BINARY, BLOB_BASE, BLOCK, BODY, BOTH, BOUND, BULK, BYTE</td>
</tr>
<tr>
<td>C</td>
<td>C, CALL, CALLING, CASCADE, CHAR, CHAR_BASE, CHARACTER, CHARSETID, CHARSET, CLOB_BASE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONTINUE, CONVERT, COUNT, CURSOR, CUSTOMDATUM</td>
</tr>
<tr>
<td>D</td>
<td>DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DETERMINISTIC, DOUBLE, DURATION</td>
</tr>
<tr>
<td>E</td>
<td>ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPT, EXCEPTIONS, EXECUTE, EXIT, EXTERNAL</td>
</tr>
<tr>
<td>F</td>
<td>FINAL, FIXED, FLOAT, FORALL, FORCE, FUNCTION</td>
</tr>
<tr>
<td>G</td>
<td>GENERAL</td>
</tr>
<tr>
<td>H</td>
<td>HASH, HEAP, HIDDEN, HOUR</td>
</tr>
<tr>
<td>I</td>
<td>IMMEDIATE, INCLUDING, INDICATOR, INDICES, INFINITE, INSTANTIABLE, INT, INTERFACE, INTERVAL, INVALIDATE, ISOLATION</td>
</tr>
<tr>
<td>J</td>
<td>JAVA</td>
</tr>
<tr>
<td>L</td>
<td>LANGUAGE, LARGE, LEADING, LENGTH, LEVEL, LIBRARY, LIKE2, LIKE4, LIKEC, LIMIT, LIMITED, LOCAL, LONG, LOOP</td>
</tr>
<tr>
<td>M</td>
<td>MAP, MAX, MAXLEN, MEMBER, MERGE, MIN, MINUTE, MOD, MODIFY, MONTH, MULTISET</td>
</tr>
<tr>
<td>N</td>
<td>NAME, NAN, NATIONAL, NATIVE, NCHAR, NEW, NOCOPY, NUMBER_BASE</td>
</tr>
<tr>
<td>O</td>
<td>OBJECT, OCICOLL, OCIDATETIME, OCIDATE, OCIDURATION, OCINTERVAL, OCILOBLOCATOR, OCINUMBER, OCRAW, OCIREFCURSOR, OCIREF, OCIROWID, OCISTRING, OCITYPE, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLIVARY, OTHERS, OUT, OVERRIDING</td>
</tr>
<tr>
<td>P</td>
<td>PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARTITION, PASCAL, PIPE, PIPELINED, PRAGMA, PRECISION, PRIVATE</td>
</tr>
<tr>
<td>R</td>
<td>RAISE, RANGE, RAW, READ, RECORD, REF, REFERENCE, RELIES_ON, REM, REMAINDER, RENAME, RESULT, RESULT_CACHE, RETURN, RETURNING, REVERSE, ROLLBACK, ROW</td>
</tr>
<tr>
<td>S</td>
<td>SAMPLE, SAVE, SAVEPOINT, SB1, SB2, SB4, SECOND, SEGMENT, SELF, SEPARATE, SEQUENCE, SERIALIZABLE, SET, SHORT, SIZE_T, SOME, SPARSE, SQCODE, SQLDATA, SQLNAME, SQLSTATE, STANDARD, STATIC, STDDEV, STORED, STRING, STRUCT, STYLE, SUBMULTISET, SUBPARTITION, SUBSTITUTABLE, SUBTYPE, SUM, SYNONYM</td>
</tr>
<tr>
<td>T</td>
<td>TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABBR, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TRAILING, TRANSACTION, TRANSACTIONAL, TRUSTED, TYPE</td>
</tr>
<tr>
<td>U</td>
<td>UB1, UB2, UB4, UNDER, UNSIGNED, UNTRUSTED, USE, USING</td>
</tr>
<tr>
<td>V</td>
<td>VALIST, VALUE, VARIABLE, VARIANCE, VARRAY, VARYING, VOID</td>
</tr>
<tr>
<td>W</td>
<td>WHILE, WORK, WRAPPED, WRITE</td>
</tr>
<tr>
<td>Y</td>
<td>YEAR</td>
</tr>
<tr>
<td>Z</td>
<td>ZONE</td>
</tr>
</tbody>
</table>
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
|| concatenation operator, 2-28
. item separator, 2-3
<< label delimiter, 2-3
.. range operator, 2-3, 4-13
@ remote access indicator, 2-3, 2-19
-- single-line comment delimiter, 2-3
; statement terminator, 2-3, 13-15
- subtraction/negation operator, 2-3

A
ACCESS_INTO_NULL exception, 11-4
actual parameters, 6-22
address
   REF CURSOR, 6-23
advantages
   PL/SQL, 1-1
AFTER clause
   of CREATE TRIGGER, 14-56
AFTER triggers, 14-56
auditing and, 9-32, 9-34
correlation names and, 9-20
specifying, 9-7
aggregate assignment, 2-16
aggregate functions
   and PL/SQL, 6-3
   user-defined, 14-35
aliases
   using with a select list, 2-17
aliasing
   bulk binds and, 12-22
   for expression values in a cursor FOR loop, 6-19
   subprogram parameters and, 8-25
ALL row operator, 6-3, 6-7
ALTER FUNCTION statement, 14-3
ALTER PACKAGE statement, 14-6
ALTER PROCEDURE statement, 14-9
ALTER statements, 14-1
   triggers on, 14-59
ALTER TABLE statement
   DISABLE ALL TRIGGERS clause, 9-29
   ENABLE ALL TRIGGERS clause, 9-29
ALTER TRIGGER statement, 14-12
   DISABLE clause, 9-29
   ENABLE clause, 9-29
ALTER TYPE statement, 14-15
analytic functions
   user-defined, 14-35
anonymous block
   definition of, 1-5
ANSI/ISO SQL standard, 6-1
apostrophes, 2-8
architecture
   PL/SQL, 1-24
ARRAY
   VARYING, 5-7
arrays
   associative, 5-2
   characteristic of, 5-2
   globalization and, 5-3
   in other languages
      simulating with varrays, 5-5
      multidimensional, 5-19
      variable-size (varrays)
         characteristics of, 5-2
AS EXTERNAL clause
   of CREATE FUNCTION, 14-49
   of CREATE TYPE BODY, 14-88
AS OBJECT clause
   of CREATE TYPE, 14-71
AS TABLE clause
   of CREATE TYPE, 14-78
AS VARRAY clause
   of CREATE TYPE, 14-77
assignment operator, 1-7
assignment statement
   links to examples, 13-5
   syntax, 13-3
assignments
   aggregate, 2-16
collection, 5-13
field, 5-34
IN OUT parameters, 1-8
records, 5-34
variables, 1-7
associative arrays, 5-2
characteristic of, 5-2
compared to nested tables, 5-6
globalization and, 5-3
syntax, 13-22
asynchronous operations, 10-10
attributes
%ROWTYPE, 1-10, 2-15
%TYPE, 1-10
effective cursors, 6-13
of user-defined types
mapping to Java fields, 14-73
auditing
triggers and, 9-31
AUTHID clause
of ALTER TYPE, 14-23
AUTHID CURRENT_USER clause
of CREATE FUNCTION, 14-33
of CREATE PACKAGE, 14-40
of CREATE PROCEDURE, 14-48
of CREATE TYPE, 14-22, 14-71
AUTHID DEFINER clause
of CREATE FUNCTION, 14-33
of CREATE PACKAGE, 14-40
of CREATE PROCEDURE, 14-48
of CREATE TYPE, 14-22, 14-71
AUTHID property, 8-18
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
bags
simulating with nested tables, 5-5
basic loops, 4-9
BEFORE clause
of CREATE TRIGGER, 14-55
BEFORE triggers, 14-55
complex security authorizations, 9-41
correlation names and, 9-20
derived column values, 9-42
specifying, 9-7
BEGIN
start of executable PL/SQL block, 13-13
syntax, 13-13
BETWEEN clause
FOR ALL, 13-72
BETWEEN comparison operator, 2-37
expressions, 13-62
BFILE data type, 3-23
BINARY_DOUBLE data type, 3-5
BINARY_FLOAT and BINARY_DOUBLE data types
for computation-intensive programs, 12-27
BINARY_FLOAT data type, 3-5
BINARY_INTEGER data type
See PLS_INTEGER data type
bind arguments
avoiding SQL injection with, 7-14
bind variables, 1-9
binding
bulk, 12-10
variables, 12-9
BLOB data type, 3-23
block
anonymous
definition of, 1-5
blocks
links to examples, 13-15
PL/SQL
syntax, 13-8
BODY
with SQL CREATE PACKAGE statement, 10-1
body
cursor, 10-12
package, 10-5
BODY clause
of ALTER PACKAGE, 14-7
Boolean
assigning values, 2-27
expressions, 2-38
literals, 2-8
BOOLEAN data type, 3-15
bounded collections, 5-2
bulk
fetches, 12-19
returns, 12-21
bulk binding, 12-10
limitations, 12-10
BULK clause
with COLLECT, 12-17
BULK COLLECT clause, 12-17
checking whether no results are returned, 12-18
FETCH, 13-69
retrieving DML results, 12-21
retrieving query results with, 12-17
returning multiple rows, 6-17
SELECT INTO, 13-118
using LIMIT clause, 12-18, 12-20
using ROWNUM pseudocolumn, 12-18
using SAMPLE clause, 12-18
Index-2
using with FORALL statement, 12-21
BULK COLLECT INTO clause
  in EXECUTE IMMEDIATE statement, 13-48
  in RETURNING INTO clause, 13-112
bulk SQL
  using to reduce loop overhead, 12-9
BULK_EXCEPTIONS cursor attribute
  ERROR_CODE field, 12-16
  ERROR_INDEX field, 12-16
  example, 12-17
  handling FORALL exceptions, 12-16
  using ERROR_CODE field with SQLERRM, 12-16
BULK_ROWCOUNT cursor attribute
  affected by FORALL, 12-14
by-reference parameter passing, 8-25
by-value parameter passing, 8-25

C

C clause
  of CREATE TYPE, 14-75
  of CREATE TYPE BODY, 14-87
C method
  mapping to an object type, 14-75
call spec: See call specifications
call specification, 10-2
call specifications
  in procedures, 14-46
  of CREATE PROCEDURE, 14-49
  of CREATE TYPE, 14-75
  of CREATE TYPE BODY, 14-87
call stack
  AUTHID property and, 8-19
  DR and IR units and, 8-19
calls
  inter-language, 8-23
  resolving subprogram, 8-16
  subprograms, 8-11
carriage returns, 2-2
CASE expressions, 2-40
  overview, 1-14
case sensitivity
  in identifiers, 2-4
  string literal, 2-8
CASE statement
  links to examples, 13-18
  searched, 4-6
  syntax, 13-17
  using, 4-5
CASE_NOT_FOUND exception, 11-4
CHAR data type, 3-8
  differences with VARCHAR2, 3-9
character literals, 2-7
character sets
  PL/SQL, 2-1
CHARACTER subtype, 3-9
character values
  comparing, 3-10
CHECK constraint
  triggers and, 9-36, 9-40
clauses
  BULK COLLECT, 12-17
  LIMIT, 12-20
CLOB data type, 3-23
CLOSE statement
  disables cursor, 6-13
  disabling cursor variable
  closing, 6-29
  links to examples, 13-20
  syntax, 13-20
collating sequence, 2-39
COLLECT clause
  with BULK, 12-17
collection exceptions
  when raised, 5-30
collection methods, 5-20
  syntax, 13-27
COLLECTION_IS_NULL exception, 11-4
collections, 5-1
  allowed subscript ranges, 5-12
  applying methods to parameters, 5-28
  assigning, 5-13
  avoiding exceptions, 5-29
  bounded, 5-2
  bulk binding, 5-38, 12-9
  choosing the type to use, 5-5
  comparing, 5-17
  constructors, 5-10
  declaring variables, 5-8
  defining types, 5-7
  DELETE method, 5-27
dense, 5-2
element types, 5-7
EXISTS method, 5-21
EXTEND method, 5-25
initializing, 5-10
links to examples, 13-25, 13-30
multidimensional, 5-19
NEXT method, 5-24
operators to transform nested tables, 5-14
overview, 1-11
PRIOR method, 5-24
referencing, 5-10
referencing elements, 5-12
scope, 5-7
sparse, 5-2
syntax, 13-22
testing for null, 5-17
three types of, 5-1
TRIM method, 5-26
unbounded, 5-2
column aliases
  expression values in a cursor loop, 6-19
  when needed, 2-17
columns
  accessing in triggers, 9-20
  generating derived values with triggers, 9-42
  listing in an UPDATE trigger, 9-7, 9-22
COMMENT clause
  using with transactions, 6-34
comments
  in PL/SQL, 2-9
  links to examples, 13-31
  syntax, 13-31
COMMIT statement, 6-33
comparison functions
  MAP, 14-87
  ORDER, 14-87
comparison operators, 6-6
comparisons
  of character values, 3-10
  of expressions, 2-38
  of null collections, 5-17
  operators, 2-34
  PL/SQL, 2-28
  with NULLs, 2-42
COMPILE clause
  of ALTER PACKAGE, 14-7
  of ALTER PROCEDURE, 14-10
  of ALTER TRIGGER, 14-13
  of ALTER TYPE, 14-20
compiler parameters
  and REUSE SETTINGS clause, 1-27
  PL/SQL, 1-25
compiler switches
  dropping and preserving, 14-4, 14-8, 14-10, 14-14, 14-21
compiling
  conditional, 2-48
  composite types, 5-1
  composite variables, 5-1
  Compound triggers, 9-13
compound triggers
  creating, 14-58
concatenation operator, 2-28
  treatment of nulls, 2-44
conditional compilation, 2-48
  availability for previous Oracle database releases, 2-48
  control tokens, 2-48
  examples, 2-54
  inquiry directives, 2-49
  limitations, 2-55
  PLSQL_LINE flag, 2-50
  PLSQL_UNIT flag, 2-50
  restrictions, 2-55
  static constants, 2-52
  using static expressions with, 2-50
  using with DBMS_DB_VERSION, 2-53
  using with DBMS_PREPROCESSOR, 2-55
conditional control, 4-2
conditional predicates
  trigger bodies, 9-18, 9-22
conditional statement
  guidelines, 4-7
CONSTANT
  for declaring constants, 1-9, 2-11
constants
  declaring, 1-9, 2-10, 2-11
  links to examples, 13-34, 13-132
  static, 2-52
  syntax, 13-32, 13-130
  understanding PL/SQL, 1-6
  constraining tables, 9-25
  constraints
    NOT NULL, 2-12
    triggers and, 9-3, 9-35
    constructor methods
      and object types, 14-71
    constructors
      collection, 5-10
      defining for an object type, 14-76
      user-defined, 14-76
  context
    transactions, 6-43
CONTINUE statement
  links to examples, 13-35
  syntax, 13-35
CONTINUE-WHEN statement, 1-16
control structures
  conditional, 4-2
  overview of PL/SQL, 4-1
  sequential, 4-20
  understanding, 1-13
conventions
  PL/SQL naming, 2-19
  conversions
    data type, 3-28
    correlated subqueries, 6-20
    correlation names, 9-13
      NEW, 9-20
      OLD, 9-20
      when preceded by a colon, 9-20
COUNT collection method, 5-21
COUNT method
  collections, 13-27
CREATE
  with PROCEDURE statement, 1-18
  CREATE FUNCTION statement, 14-29
  CREATE PACKAGE BODY statement, 14-42
  CREATE PACKAGE statement, 14-39
  CREATE PROCEDURE statement, 1-18, 14-46
  CREATE statement
    packages, 10-1
  CREATE statements, 14-1
    triggers on, 14-59
  CREATE TRIGGER statement, 9-5, 14-51
    REFERENCING option, 9-21
  CREATE TYPE BODY statement, 14-83
  CREATE TYPE statement, 14-65
  creating
    packages, 10-1
    procedures, 1-18
  CURRENT OF clause
    with UPDATE, 6-38
  CURRENT_USER
    value of AUTHID property, 8-18
CURRVAL
  pseudocolumn, 6-4
cursor attributes
%BULK_EXCEPTIONS, 12-16
%BULK_ROWCOUNT, 12-14
%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

syntax, 13-37
links to examples, 13-38
native dynamic SQL and, 7-2
SQL, 6-8
values of, 6-15
cursor declarations
links to examples, 13-56
syntax, 13-53
cursor expressions
REF CURSORS, 6-32
restrictions, 6-32
using, 6-31
cursor FOR loops
passing parameters to, 6-21
cursor subqueries
using, 6-31
cursor variables, 6-22
advantages of, 6-23
as parameters to table functions, 12-38
avoiding errors with, 6-30
closing, 6-29
declaring, 6-23
defining, 6-23
fetching from, 6-28
links to examples, 13-42
opening, 6-25
passing as parameters, 6-24
reducing network traffic, 6-29
restrictions, 6-30
syntax, 13-39
using as a host variable, 6-27
CURSOR_ALREADY_OPEN exception, 11-4
cursors
advantages of using cursor variables, 6-23
attributes of explicit, 6-13
attributes of SQL, 6-8
closing explicit, 6-13
declaring explicit, 6-10
definition, 1-10
explicit, 1-10, 6-9
explicit FOR loops, 6-18
expressions, 6-31
fetching from, 6-11
guidelines for implicit, 6-9
implicit, 1-10
opening explicit, 6-11
packaged, 10-12
parameterized, 6-21
REF CURSOR variables, 6-22
RETURN clause, 10-12
scope rules for explicit, 6-10
SYS_REFCURSOR type, 12-38

variables, 6-22
CustomDatum Java storage format, 14-72

D

data abstraction
understanding PL/SQL, 1-9
data definition language (DDL)
events and triggers, 14-59
data manipulation language
triggers and, 9-2
data manipulation language (DML)
operations
and triggers, 14-57
data type conversion
SQL injection and, 7-12
data types
BFILE, 3-23
BLOB, 3-23
BOOLEAN, 3-15
CHAR, 3-8
CLOB, 3-23
DATE, 3-16
explicit conversion, 3-28
implicit conversion, 3-29
INTERVAL DAY TO SECOND, 3-20
INTERVAL YEAR TO MONTH, 3-20
LONG, 3-14
national character, 3-12
NCHAR, 3-13, 3-14
NCLOB, 3-23
NUMBER, 3-6
PL/SQL
See PL/SQL data types
RAW, 3-12
REF CURSOR, 6-23
ROWID, 3-14
TABLE, 5-7
TIMESTAMP, 3-17
TIMESTAMP WITH LOCAL TIME ZONE, 3-19
TIMESTAMP WITH TIME ZONE, 3-18
UROWID, 3-14
VARRAY, 5-7
database character set, 2-1
database events
attributes, 9-46
tracking, 9-44
Database Resident Connection Pool, 10-11
database triggers, 1-20
autonomous, 6-45
database triggers. See triggers
databases
events
and triggers, 14-60
auditing, 14-60
transparent logging of, 14-60
DATE data type, 3-16
datetime
arithmetic, 3-16
data types, 3-15
DAY  
data type field, 3-16  

DB_ROLE_CHANGE system manager event, 9-50  

DBMS_ALERT package, 10-10  

DBMS_ASSERT package, 7-15  

DBMS_CONNECTION_CLASS package, 10-11  

DBMS_DB_VERSION package  
using with conditional compilation, 2-53  

DBMS_OUTPUT package  
displaying output, 1-6  

displaying output from PL/SQL, 10-10  

DBMS_PIPE package, 10-11  

DBMS_PREPROCESSOR package  
using with conditional compilation, 2-55  

DBMS_PROFILE package  
gathering statistics for tuning, 12-8  

DBMS_SQL package, 7-6  

upgrade to dynamic SQL, 12-28  

DBMS_SQL_TO_NUMBER function, 7-6  

DBMS_SQL_TO_REFCURSOR function, 7-6  

DBMS_TRACE package  
tracing code for tuning, 12-9  

DBMS_WARNING package  
controlling warning messages in PL/SQL, 11-21  

dbmsupbin.sql script  
interpreted compilation, 12-32  

dbmsupgnv.sql script  
for PL/SQL native compilation, 12-33  

deadlocks  
how handled by PL/SQL, 6-36  

DEBUG clause  
of ALTER FUNCTION, 14-4  
of ALTER PACKAGE, 14-7  
of ALTER PROCEDURE, 14-10  
of ALTER TRIGGER, 14-13  
of ALTER TYPE, 14-20  

debugging  
triggers, 9-29  

DEC  
NUMBER subtype, 3-7  

DECIMAL  
NUMBER subtype, 3-7  

declarations  
collection, 5-8  

constants, 1-9, 2-11  
cursor variables, 6-23  
exceptions in PL/SQL, 11-6  
explicit cursor, 6-10  
PL/SQL functions, 1-18  
PL/SQL procedures, 1-18  
PL/SQL subprograms, 1-18  
restrictions, 2-18  
using %ROWTYPE, 2-15  
using DEFAULT, 2-11  
using NOT NULL constraint, 2-12  
variables, 1-7, 2-10  

DECLARE  
start of declarative part of a PL/SQL block, 13-14  
syntax, 13-14  

DECODE function  
treatment of nulls, 2-45  

DEFAULT keyword  
for assignments, 2-11  

DEFAULT option  
RESTRICT_REFERENCES, 13-107  
default parameter values, 8-9  

DEFINE  
limitations of use with wrap utility, A-4  

DEFINER value of AUTHID property, 8-18  
definer's rights functions, 14-33  
definer's rights units  
See DR units  

DELETE method  
collections, 5-27, 13-28  

DELETE statement  
column values and triggers, 9-20  

triggers for referential integrity, 9-37, 9-38  

triggers on, 14-57  

delimiters, 2-3  
dense collections, 5-2  
dense nested tables, 5-4  
dependencies  
in stored triggers, 9-28  
schema objects  

trigger management, 9-25  

DETERMINISTIC clause  
of CREATE FUNCTION, 14-33  

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, 3-7  
disabled trigger  
definition, 9-2  

disabling  
triggers, 9-2  
displaying output  
DBMS_OUTPUT package, 1-6  
setting SERVEROUTPUT, 10-10  

DISTINCT row operator, 6-3, 6-7  
distributed databases  
triggers and, 9-25  
dot notation, 1-10, B-3  
for collection methods, 5-21  
for global variables, 4-18  
for package contents, 10-4  

DOUBLE PRECISION  
NUMBER subtype, 3-7  

DR units  
call stack and, 8-19  
dynamic SQL statements and, 8-19  
name resolution and, 8-18  
privilege checking and, 8-18  
static SQL statements and, 8-19  

DROP PACKAGE BODY statement, 14-91  
DROP statements, 14-2  

triggers on, 14-59  

DROP TRIGGER statement, 9-29
dropping triggers, 9-29
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, 12-27
dynamic SQL statements
  AUTHID property and, 8-19

E
  element types
    collection, 5-7
  ELSE clause
    using, 4-3
  ELSIF clause
    using, 4-4
  ENABLE clause
    of ALTER TRIGGER, 14-13
    enabled trigger
      definition, 9-2
    enabling
      triggers, 9-2
END
  end of a PL/SQL block, 13-14
  syntax, 13-14
END IF
  end of IF statement, 4-2
END LOOP
  end of LOOP statement, 4-13
error handling
  in PL/SQL, 11-1
  overview, 1-5
error messages
  maximum length, 11-15
ERROR_CODE
  BULK_EXCEPTIONS cursor attribute field, 12-16
  using with SQLERRM, 12-16
ERROR_INDEX
  BULK_EXCEPTIONS cursor attribute field, 12-16
evaluation
  short-circuit, 2-34
  event attribute functions, 9-46
  event publication, 9-45 to 9-46
  triggering, 9-45
events
  attribute, 9-46
  tracking, 9-44
EXCEPTION
  exception-handling part of a block, 13-14
  syntax in PL/SQL block, 13-14
exception definition
  syntax, 13-44, 13-45
exception handlers
  OTHERS handler, 11-2
  overview, 1-5
  using RAISE statement in, 11-13
  WHEN clause, 11-13
EXCEPTION_INIT pragma
  links to examples, 13-43
  syntax, 13-43
  using with RAISE_APPLICATION_ERROR, 11-9
  with exceptions, 11-7
exceptions
  advantages of PL/SQL, 11-3
  branching with GOTO, 11-15
  catching unhandled in PL/SQL, 11-16
  continuing after an exception is raised, 11-17
  controlling warning messages, 11-20
  declaring in PL/SQL, 11-6
  definition, 13-44, 13-45
  during trigger execution, 9-22
  handling in PL/SQL, 11-1
  links to examples, 13-44, 13-46
  list of predefined in PL/SQL, 11-4
  locator variables to identify exception locations, 11-19
  OTHERS handler in PL/SQL, 11-14
  PL/SQL compile-time warnings, 11-19
  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, 11-15
  raising in PL/SQL, 11-9
  raising predefined explicitly, 11-10
  raising with RAISE statement, 11-9
  redeclaring predefined in PL/SQL, 11-9
  reraising in PL/SQL, 11-13
  retrying a transaction after, 11-18
  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 the DBMS_WARNING package, 11-21
  using WHEN and OR, 11-14
  WHEN clause, 11-13
EXECUTE IMMEDIATE statement, 7-2
  links to examples, 13-49
  syntax, 13-47
EXISTS method
  collections, 5-21, 13-28
EXIT statement
  early exit of LOOP, 4-19
  links to examples, 13-51
  syntax, 13-51
  using, 4-9, 4-10
EXIT-WHEN statement, 1-16
  using, 4-10, 4-11
  explicit cursors, 6-9
  explicit data type conversion, 3-28
  explicit declarations
    cursor FOR loop record, 6-18
  explicit format models
  avoiding SQL injection
Index-8

with, 7-17
expressions
  as default parameter values, 8-10
    in cursors, 6-22
  Boolean, 2-38
  CASE, 2-40
  examples, 13-67
  PL/SQL, 2-28
  static, 2-50
  syntax, 13-57
EXTEND method
  collections, 5-25, 13-28
external
  routines, 8-23
  subprograms, 8-23
external functions, 14-29, 14-46
external procedures, 14-46

F
FALSE value, 2-8
FETCH statement
  links to examples, 13-70
  syntax, 13-68
  using explicit cursors, 6-11
  with cursor variable, 6-28
fetching
  across commits, 6-39
  bulk, 12-19
  file I/O, 10-11
FINAL clause
  of CREATE TYPE, 14-73, 14-74
FIRST collection method, 5-23, 13-28
FIRST method
  collections, 13-28
FLOAT
  NUMBER subtype, 3-7
FOR EACH ROW clause, 9-12
  of CREATE TRIGGER, 14-58
FOR loops
  explicit cursors, 6-18
  nested, 4-18
FOR UPDATE clause, 6-11
  when to use, 6-38
FORALL statement
  links to examples, 13-74
  syntax, 13-72
  using, 12-10
  using to improve performance, 12-10
  using with BULK COLLECT clause, 12-21
  with rollbacks, 12-14
FORCE clause
  of DROP TYPE, 14-96
FOR-LOOP statement
  syntax, 13-90
  using, 4-13
  formal parameters, 6-22
  format models
    explicit avoiding SQL injection with, 7-17
    forward
data type of return value, 13-78, 14-32
declaration, 13-75
data type of return value, 13-78, 14-32
declaration, 13-75
examples, 14-36
executing
  from parallel query processes, 14-34
  external, 14-29, 14-46
  in PL/SQL, 8-1
  invoking, 8-2
  links to examples, 13-78
partitioning
  among parallel query processes, 14-34
  pipelined, 12-34
privileges executed with, 14-22, 14-71
recompiling invalid, 14-3
re-creating, 14-32
removing from the database, 14-89
RETURN statement, 8-4
returning collections, 14-34
returning results iteratively, 14-34
schema executed in, 14-22, 14-71
specifying schema and user privileges for, 14-33
SQL in PL/SQL, 2-47
stored, 14-29
table, 12-34, 14-34
user-defined
  aggregate, 14-35
  using a saved copy, 14-33

G
global identifiers, 2-23
globalization
  associative arrays and, 5-3
GOTO statement
  branching into or out of exception handler, 11-15
  label, 4-20
  links to examples, 13-79
  overview, 1-17
  syntax, 13-79
  using, 4-20
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, 11-15
        using OTHERS handler, 11-14
handling of nulls, 2-42
hash tables
    simulating with associative arrays, 5-5
hiding PL/SQL source code
    PL/SQL source code
host arrays
    bulk binds, 12-22
HOUR
    data type field, 3-16
HTF package, 10-11
HTTP package, 10-11
hypertext markup language (HTML), 10-11
hypertext transfer protocol (HTTP), 1-3
    UTL_HTTP package, 10-11

identifiers
    global, 2-23
    local, 2-23
    quoted, 2-5
    scope and visibility of, 2-22
    syntax and semantics of, 2-4
IF statement, 4-2
    ELSE clause, 4-3
    links to examples, 13-81, 13-83
    syntax, 13-80
    using, 4-2
IF-THEN statement
    using, 4-2
IF-THEN-ELSE statement
    overview, 1-14
    using, 4-3
IF-THEN-ELSIF statement
    using, 4-4
implicit cursors
    guidelines, 6-9
    See SQL cursors
implicit data type conversion, 3-29
implicit data type conversions
    performance, 12-5
implicit declarations
    FOR loop counter, 4-18
    IN comparison operator, 2-37
    IN OUT parameter mode
        subprograms, 8-9
    IN parameter mode
        subprograms, 8-8
    incomplete object types, 14-65
    creating, 14-65
INDEX BY
    collection definition, 13-24

index-by tables
    See associative arrays
INDICES OF clause
    FOR ALL, 13-72
    with FOR ALL, 12-10
infinite loops, 4-9
initialization
    collections, 5-10
    package, 10-6
    using DEFAULT, 2-11
    variable, 2-26
initialization parameters
    PL/SQL compilation, 1-25
injection, SQL, 7-11
inline LOB locators, 3-22
INLINE pragma
    syntax, 13-82
    Inlineing subprograms, 12-1
    input, 1-6
    input-output packages, 1-6
INSERT statement
    column values and triggers, 9-20
    triggers on, 14-57
    with a record variable, 5-36
    instance_num event attribute, 9-47
INSTANTIABLE clause
    of CREATE TYPE, 14-73
INSTEAD OF clause
    of CREATE TRIGGER, 14-56
    INSTEAD OF triggers, 9-8, 14-56
    on nested table view columns, 9-21
INT
    NUMBER subtype, 3-7
    INTEGER
        NUMBER subtype, 3-7
    interpreted compilation
        dbmsupbin.sql script, 12-32
        recompiling all PL/SQL modules, 12-32
INTERSECT set operator, 6-7
interval
    arithmetic, 3-21
    INTERVAL DAY TO SECOND data type, 3-20
    INTERVAL YEAR TO MONTH data type, 3-20
    intervals
        data types, 3-15
INTO
    SELECT INTO statement, 13-116
    INTO clause
        with FETCH statement, 6-29
    INTO list
        with using explicit cursors, 6-11
    INVALID_CURSOR exception, 11-5
    INVALID_NUMBER exception, 11-5
invoker’s rights
    altering for an object type, 14-22
    defining for a function, 14-33
    defining for a package, 14-39
    defining for a procedure, 14-47
    defining for an object type, 14-71
invoker’s rights functions
defining, 14-33
invoker’s rights subprograms
name resolution in, 8-20
invoker’s rights units
See IR units
invoking Java stored procedures, 8-23
IR units
call stack and, 8-19
dynamic SQL statements and, 8-19
name resolution and, 8-18
privilege checking and, 8-18
static SQL statements and, 8-19
IS NULL comparison operator, 2-35
expressions, 13-64
is_alter_column event attribute, 9-47
ISOPEN cursor attribute
explicit, 6-14
implicit, 6-8

J
JAVA
use for invoking external subprograms, 8-24
Java
call specs, 8-24
methods
return type of, 14-74
storage formats
CustomDatum, 14-72
SQLData, 14-72
JAVA clause
of CREATE TYPE, 14-75
of CREATE TYPE BODY, 14-87
Java methods
mapping to an object type, 14-75
Java stored procedures
invoking from PL/SQL, 8-23

K
keywords, 2-5
use in PL/SQL, 2-5
keywords in PL/SQL, D-1

L
labels
block structure, 13-14
exiting loops, 4-12
GOTO statement, 4-20
loops, 4-12
syntax, 13-14
LANGUAGE
use for invoking external subprograms, 8-24
LANGUAGE clause
of CREATE PROCEDURE, 14-49
of CREATE TYPE, 14-75
of CREATE TYPE BODY, 14-87
language elements
of PL/SQL, 13-1
large object (LOB) data types, 3-22
LAST collection method, 5-23, 13-28
LAST method
collections, 13-29
LEVEL
 pseudocolumn, 6-5
lexical units
PL/SQL, 2-1
LIKE comparison operator, 2-35
expressions, 13-64
LIMIT clause
FETCH, 13-69
using to limit rows for a Bulk FETCH operation, 12-20
LIMIT collection method, 5-22
LIMIT method
collections, 13-29
limitations
bulk binding, 12-10
of PL/SQL programs, C-1
PL/SQL compiler, C-1
limits
on PL/SQL programs, C-1
literals
Boolean, 2-8
datetime, 2-8
examples, 13-87
numeric, 2-6
numeric data types, 2-6
syntax, 13-85
types of PL/SQL, 2-6
LOB (large object) data types, 3-22
use in triggers, 9-20
LOB locators, 3-22
local identifiers, 2-23
locator variables
used with exceptions, 11-19
LOCK TABLE statement
locking a table, 6-39
locks
modes, 6-33
overriding, 6-37
transaction processing, 6-32
using FOR UPDATE clause, 6-38
logical operators, 2-30
logical rowids, 3-14
LOGIN_DENIED exception, 11-5
LOGOFF database event
triggers on, 14-60
LOGON database event
triggers on, 14-60
LONG data type, 3-14
maximum length, 3-14
use in triggers, 9-25
LOOP statement, 4-8
links to examples, 13-92
overview, 1-15
syntax, 13-88
using, 4-9
loops
dynamic ranges, 4-16
exiting using labels, 4-12
implicit declaration of counter, 4-18
iteration, 4-15
labels, 4-12
reversing the counter, 4-14
scope of counter, 4-17

M

MAP MEMBER clause
of ALTER TYPE, 14-22
of CREATE TYPE, 14-87
MAP methods
defining for a type, 14-76
specifying, 14-22
maximum precision, 3-7
maximum size
CHAR value, 3-8
LONG value, 3-14
Oracle error message, 11-15
RAW value, 3-12
MEMBER clause
of ALTER TYPE, 14-21
of CREATE TYPE, 14-73
membership test, 2-37
memory
avoid excessive overhead, 12-7
Method 4, 7-6
methods
collection, 5-20
overriding a method a supertype, 14-74
preventing overriding in subtypes, 14-74
static, 14-73
without implementation, 14-74
MINUS set operator, 6-7
MINUTE
data type field, 3-16
modularity
packages, 10-3
MONTH
data type field, 3-16
multidimensional collections, 5-19
multiline comments, 2-10
multiple-row queries
dynamic, 7-4
MULTISET EXCEPT operator, 5-14
MULTISET INTERSECT operator, 5-14
MULTISET UNION operator, 5-14
mutating table
definition, 9-25
mutating tables
trigger restrictions, 9-25

N
NAME
for invoking external subprograms, 8-24
NAME parameter
transactions, 6-37
name resolution, 2-20
AUTHID property and, 8-18
differences between PL/SQL and SQL, B-4
DR units and, 8-18
global and local variables, B-1
inner capture in DML statements, B-5
IR units and, 8-18
overriding in IR subprograms, 8-20
qualified names and dot notation, B-2
qualifying references to attributes and
methods, B-6
understanding, B-1
understanding capture, B-4
names
explicit cursor, 6-10
qualified, 2-19
savepoint, 6-35
variable, 2-19
naming conventions
PL/SQL, 2-19
national character data types, 3-12
national character set, 2-1
native compilation
dbmsupp.sql script, 12-33
dependencies, 12-31
how it works, 12-31
invalidation, 12-31
modifying databases for, 12-32
revalidation, 12-31
setting up databases, 12-31
utlr.sql script, 12-33
native dynamic SQL, 7-2
NATURAL
BINARY_INTEGER subtype, 3-3
NATURAL
BINARY_INTEGER subtype, 3-3
NCHAR data type, 3-13, 3-14
NCLOB data type, 3-23
nested cursors
using, 6-31
NESTED TABLE clause
of CREATE TRIGGER, 14-58
nested tables, 5-4
characteristics of, 5-2
compared to associative arrays, 5-6
compared to varrays, 5-6
creating, 14-65
dropping the body of, 14-97
dropping the specification of, 14-95
modifying, 14-25
of scalar types, 14-25
syntax, 13-22
transforming with operators, 5-14
update in a view, 14-56
nesting
FOR loops, 4-18
record, 5-33
NEW correlation name, 9-20
new features, xxxv
NEXT method
collections, 5-24, 13-29
NEXTVAL
pseudocolumn, 6-4
NLS parameters
SQL injection and, 7-12
NLS_COMP initialization parameter
associative arrays and, 5-3
NLS_LENGTH_SEMANTICS initialization parameter
setting with ALTER SYSTEM, 14-4
NLS_SORT initialization parameter
associative arrays and, 5-3
NO_DATA_FOUND exception, 11-5
NOCOPY compiler hint
for tuning, 12-28
restrictions on, 12-29
NOT FINAL clause
of CREATE TYPE, 14-73
NOT INSTANTIABLE clause
of CREATE TYPE, 14-73, 14-74
NOT logical operator
treatment of nulls, 2-43
NOT NULL
declaration, 13-33, 13-131
NOT NULL constraint
restriction on explicit cursors, 6-10
using in collection declaration, 5-10
using in variable declaration, 2-12
NOT NULL option
record definition, 13-105
NOT_LOGGED_ON exception, 11-5
notation
positional and named, 8-11
NOTFOUND cursor attribute
explicit, 6-14
implicit, 6-8
NOWAIT parameter
using with FOR UPDATE, 6-38
NVL function
treatment of nulls, 2-45
null handling, 2-42
NULL statement
links to examples, 13-93
syntax, 13-93
using, 4-23
NULL value, 2-8
dynamic SQL and, 7-3
NUMBER data type, 3-6
range of literals, 2-6
range of values, 3-6
NUMERIC
NUMBER subtype, 3-7
numeric literals, 2-6
PL/SQL data types, 2-6

O

obfuscating PL/SQL source code
See wrapping PL/SQL source code
object identifiers
specifying, 14-71
object types
adding methods to, 14-23
adding new member subprograms, 14-21
allowing object instances of, 14-21
allowing subtypes, 14-73
and subtypes, 14-21
and supertypes, 14-21
bodies
creating, 14-83
re-creating, 14-86
SQL examples, 14-88
compiling the specification and body, 14-20
creating, 14-65, 14-66
defining member methods of, 14-83
disassociating statistics types from, 14-95
dropping methods from, 14-23
dropping the body of, 14-97
dropping the specification of, 14-95
function subprogram
declaring, 14-88
function subprograms, 14-21, 14-73
handling dependent types, 14-25
incomplete, 14-65
inheritance, 14-74
invalidating dependent types, 14-25
MAP methods, 14-76
ORDER methods, 14-76
overview, 1-12
privileges, 14-23
procedure subprogram
declaring, 14-88
procedure subprograms, 14-21, 14-73
root, 14-71
SQL examples, 14-78
static methods of, 14-73
subtypes, 14-71
top-level, 14-71
user-defined
creating, 14-71
using with invoker’s-rights subprograms, 8-21
values
comparing, 14-87
OBJECT_VALUE pseudocolumn, 9-22
objects. See object types or database objects
OLD correlation name, 9-20
ON DATABASE clause
of CREATE TRIGGER, 14-57
ON NESTED TABLE clause
of CREATE TRIGGER, 14-57
ON SCHEMA clause
of CREATE TRIGGER, 14-57
OPEN statement
explicit cursors, 6-11
links to examples, 13-95
syntax, 13-94
OPEN-FOR statement, 6-25
  links to examples, 13-97
  syntax, 13-96
OPEN-FOR-USING statement
  syntax, 13-96
operators
  comparison, 2-34
  logical, 2-30
  precedence, 2-28
  relational, 2-35
optimizing
  PL/SQL programs, 12-1
OR keyword
  using with EXCEPTION, 11-14
OR REPLACE clause
  of CREATE FUNCTION, 14-32
  of CREATE PACKAGE, 14-39
  of CREATE PACKAGE BODY, 14-42
  of CREATE PROCEDURE, 14-48
  of CREATE TRIGGER, 14-55
  of CREATE TYPE, 14-70
  of CREATE TYPE BODY, 14-86
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, 9-48
ora_is_drop_column event attribute, 9-48
ora_is_servererror event attribute, 9-48
ora_login_user event attribute, 9-48
ora_privileges event attribute, 9-48
ora_revokee event attribute, 9-48
ora_server_error event attribute, 9-48
ora_with_grant_option event attribute, 9-50
ORDER MEMBER clause
  of ALTER TYPE, 14-21
  of CREATE TYPE, 14-74
OVERRIDING clause
  of ALTER TYPE, 14-21
  of CREATE TYPE, 14-74
PACKAGE
  with SQL CREATE statement, 10-1
package bodies
  creating, 14-42
  re-creating, 14-42
  removing from the database, 14-91
PACKAGE BODY
  with SQL CREATE statement, 10-1
packaged cursors, 10-12
packaged procedures
  dropping, 14-93
packages
  advantages, 10-3
  avoiding run-time compilation, 14-7
  bodiless, 10-4
  body, 10-1, 10-5
  call specification, 10-2
  contents of, 10-2
  creating, 10-1, 14-39
  cursor specifications, 10-12
  cursors, 10-12
  disassociating statistics types from, 14-91
  dot notation, 10-4
  examples of features, 10-6
  global variables, 10-9
  guidelines for writing, 10-12
  hidden declarations, 10-1
  initializing, 10-6
  invoker’s rights, 14-40
  invoking subprograms, 10-5
  modularity, 10-3
  overloading subprograms, 10-9
  overview, 1-20
  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
  recompiling explicitly, 14-7
  redefining, 14-39
  referencing, 10-4
  removing from the database, 14-91
  restrictions on referencing, 10-5
  scope, 10-3
  specification, 10-1
  specifications, 10-3
  specifying schema and privileges of, 14-40
  STANDARD package, 10-9
  understanding, 10-1
  visibility of contents, 10-1
PARALLEL_ENABLE clause
  of CREATE FUNCTION, 14-34
parameter passing
  by reference, 8-25
  by value, 8-25
parameters
  actual, 6-22
  actual and formal, 8-6
  aliasing, 8-25
  cursor, 6-21
  default values, 8-9
  formal, 6-22
  IN mode, 8-8
  IN OUT mode, 8-9
  modes, 8-7
  OUT mode, 8-8
  summary of modes, 8-9
parentheses, 2-29
parse tree, 9-27
pattern matching, 2-36
performance
  avoid memory overhead, 12-7
  avoiding problems, 12-3
physical rowids, 3-14
pipe, 10-11
PIPE ROW statement
  for returning rows incrementally, 12-37
PIPELINED
  function option, 12-35, 13-77
PIPELINED clause
  of CREATE FUNCTION, 14-34
pipelined functions
  exception handling, 12-42
  fetching from results of, 12-38
  for querying a table, 12-34
  overview, 12-34
  passing data with cursor variables, 12-38
  performing DML operations inside, 12-41
  performing DML operations on, 12-41
  returning results from, 12-37
  transformation of data, 12-34
  transformations, 12-36
  writing, 12-35
pipelines
  between table functions, 12-37
  returning results from table functions, 12-37
  support collection types, 12-35
  using table functions, 12-36
  writing table functions, 12-35
pipelining
  definition, 12-35
PLS_INTEGER data type, 3-2
  overflow condition, 3-3
PL/SQL
  advantages, 1-1
  architecture, 1-24
  assigning Boolean values, 2-27
  assigning query result to variable, 2-27
  assigning values to variables, 2-26
  blocks
    syntax, 13-8
  CASE expressions, 2-40
  character sets, 2-1
  collections
    overview, 1-11
comments, 2-9
comparisons, 2-28
compiler limitations, C-1
compiler parameters, 1-25
compile-time warnings, 11-19
conditional compilation, 2-48
constants, 1-6
control structures, 1-13, 4-1
data abstraction, 1-9
declarations
  constants, 2-10
  displaying output, 10-10
  engine, 1-24
  environment, 10-9
  error handling
    overview, 1-5
  errors, 11-1
  exceptions, 11-1
  expressions, 2-28
  functions, 8-1
  lexical units, 2-1
  limitations of programs, C-1
  limits on programs, C-1
  literals, 2-6
  logical operators, 2-30
  name resolution, B-1
  naming conventions, 2-19
  new features, xxxv
  performance problems, 12-3
  portability, 1-3
  procedural aspects, 1-4
  procedures, 8-1
  profiling and tracing programs, 12-8
  querying data, 6-16
  records
    overview, 1-12
  Server Pages (PSPs), 2-57
  statements, 13-1
  subprograms, 8-1
  syntax of language elements, 13-1
  transaction processing, 6-32
  trigger bodies, 9-18, 9-20
  tuning code, 12-2
  tuning computation-intensive programs, 12-27
  tuning dynamic SQL programs, 12-27
  using NOCOPY for tuning, 12-28
  using transformation pipelines, 12-34
  variables, 1-6
  warning messages, 11-19
  Web applications, 2-56
PL/SQL compiler
  parameters, 14-4, 14-8, 14-10, 14-14, 14-21
PL/SQL data types
  predefined, 3-1
PLSQL data types
  numeric literals, 2-6
PL/SQL function result cache, 8-27
PL/SQL units
  stored
    SQL statements for, 14-1
what they are, 1-25
PLSQL_LINE flag
    use with conditional compilation, 2-50
PLSQL_OPTIMIZE_LEVEL compilation
    parameter, 12-2
    optimizing PL/SQL programs, 12-1
PLSQL_UNIT flag
    use with conditional compilation, 2-50
PLSQL_WARNINGS initialization parameter, 11-19
pointers
    REF CIRSOR, 6-23
    portability, 1-3
    BINARY_INTEGER subtype, 3-3
    POSITIVE
    BINARY_INTEGER subtype, 3-3
PRAGMA
    compiler directive with SERIALLY_REUSABLE, 13-120
PRAGMA clause
    of ALTER TYPE, 14-21
    of CREATE TYPE, 14-69, 14-75
PRAGMA RESTRICT_REFERENCES, 14-21
pragmas
    AUTONOMOUS_TRANSACTION, 6-41, 13-6
    compiler directives, 11-7
    EXCEPTION_INIT, 11-7, 13-43
    INLINE, 13-82
    RESTRICT_REFERENCES, 6-46, 8-25, 13-107
    SERIALLY_REUSABLE, 13-120
precedence of operators, 2-28
precision of digits
    specifying, 3-7
predefined exceptions
    raising explicitly, 11-10
    redeclaring, 11-9
predefined PL/SQL data types, 3-1
predicates, 6-6
PRIOR method
    collections, 5-24, 13-29
PRIOR row operator, 6-5
    privilege checking
    AUTHID property and, 8-18
    DR units and, 8-18
    IR units and, 8-18
privileges
    creating triggers, 9-4
    dropping triggers, 9-29
    recompiling triggers, 9-28
    See also privilege checking
PROCEDURE
    with CREATE statement, 1-18
procedure declaration
    syntax, 13-101
procedures
    avoid run-time compilation, 14-10
    compile explicitly, 14-10
    creating, 1-18, 14-46
declaration, 13-101
declaring
    as a Java method, 14-49
    as C functions, 14-49
    external, 14-46
    in PL/SQL, 8-1
    invalidating local objects dependent on, 14-93
    invoked by triggers, 9-25
    invoking, 8-2
    links to examples, 13-102
    privileges executed with, 14-22, 14-71
    recompiling, 14-9
    re-creating, 14-48
    removing from the database, 14-93
    schema executed in, 14-22, 14-71
    specifying schema and privileges for, 14-48
    productivity, 1-3
Profiler API
    gathering statistics for tuning, 12-8
PROGRAM_ERROR exception, 11-5
propagation
    exceptions in PL/SQL, 11-10
pseudocolumns
    CURRVAL, 6-4
    LEVEL, 6-5
    modifying views, 9-9
    NEXTVAL, 6-4
    ROWID, 6-5
    ROWNUM, 6-6
    SQL, 6-4
    UROWID, 6-5
    use in PL/SQL, 6-4
public objects
    packages, 10-9
    purity rules, 8-25
Q
qualifiers
    using subprogram names as, 2-21
queries
    multiple-row
        dynamic, 7-4
    triggers use of, 9-2
query work areas, 6-23
querying data
    BULK COLLECT clause, 6-17
    cursor FOR loop, 6-17
    implicit cursor FOR loop, 6-18
    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, 6-18
    with PL/SQL, 6-16
work areas, 6-23
quoted identifiers, 2-5
RAISE statement
- exceptions in PL/SQL, 11-9
- links to examples, 13-103
- syntax, 13-103
- using in exception handler, 11-13
raise_application_error procedure
- for raising PL/SQL exceptions, 11-8
raising an exception
- in PL/SQL, 11-9
raising exceptions
- triggers, 9-22
range operator, 4-13
RAW data type, 3-12
- maximum length, 3-12
read consistency
- triggers and, 9-2
READ ONLY parameter
- transactions, 6-37
readability
- with NULL statement, 4-23
- read-only transaction, 6-37
REAL
- NUMBER subtype, 3-7
record definition
- syntax, 13-104
records, 5-1
- %ROWTYPE, 6-18
- assigning values, 5-34
- bulk-binding collections of, 5-38
- comparing, 5-36
- declaring, 5-31
- defining, 5-31
- definition, 1-10, 13-104
- implicit declaration, 6-18
- inserting, 5-36
- links to examples, 13-106
- manipulating, 5-33
- nesting, 5-33
- overview, 1-12
- passing as parameters, 5-33
- restriction on assignments, 5-35
- restrictions on inserts and updates of, 5-38
- returning into, 5-37
- updating, 5-37
- using as function return values, 5-33
recursion
- using with PL/SQL subprograms, 8-23
REF CURSOR data type, 6-23
- cursor variables, 6-22
- defining, 6-23
- using with cursor subqueries, 6-32
REF CURSOR variables
- as parameters to table functions, 12-38
- predefined SYS_REFCURSOR type, 12-38
referencing
- collections, 5-10
REFERENCING clause
- of CREATE TRIGGER, 14-53, 14-54, 14-58
- referencing elements
- allowed subscript ranges, 5-12
REFERENCING option, 9-21
referential integrity
- self-referential constraints, 9-38
- triggers and, 9-36 to 9-39
regular expression functions
- REGEXP_LIKE, 6-11
- relational operators, 2-35
RELIES ON clause, 13-77
remote access indicator, 2-19
remote exception handling, 9-23
RENAME clause
- of ALTER TRIGGER, 14-13
REPEAT UNTIL structure
- PL/SQL equivalent, 4-13
REPLACE AS OBJECT clause
- of ALTER TYPE, 14-21
REPLACE function
- treatment of nulls, 2-46
- reraising an exception, 11-13
reserved words, 2-5
- reserved words in PL/SQL, D-1
resolution
- name, 2-20
- references to names, B-1
RESTRICT_REFERENCES pragma, 8-25
- links to examples, 13-108
- of ALTER TYPE, 14-21
- syntax, 13-107
- using with autonomous functions, 6-46
restrictions
- cursor expressions, 6-32
- cursor variables, 6-30
- overloading subprograms, 8-14
- system triggers, 9-27
result cache, 8-27
result sets, 6-11
RESULT_CACHE clause, 13-77
RETURN clause, 13-78
- cursor, 10-12
- cursor declaration, 13-40
- of CREATE FUNCTION, 14-32
- of CREATE TYPE, 14-74
- of CREATE TYPE BODY, 14-88
RETURN statement
- functions, 8-4
- links to examples, 13-109
- syntax, 13-109
return types
- REF CURSOR, 6-23
RETURNING clause
- links to examples, 13-113
- with a record variable, 5-37
RETURNING INTO clause
- syntax, 13-111
returns
- bulk, 12-21
REUSE SETTINGS clause
- of ALTER FUNCTION, 14-4
- of ALTER PACKAGE, 14-8
single-line comments, 2-9
size limit
  varrays, 5-7
SMALLINT
  NUMBER subtype, 3-7
sparse collections, 5-2
sparse nested tables, 5-4
specification
  call, 10-2
  cursor, 10-12
  package, 10-3
SPECIFICATION clause
  of ALTER PACKAGE, 14-7
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, 6-4
  static, 6-1
SQL cursor
  dynamic SQL and, 7-6
  links to examples, 13-123
  syntax, 13-122
SQL cursors
  attributes, 6-8
SQL functions in PL/SQL, 2-47
SQL injection, 7-9
SQL statements
  ALTER, 14-1
  CREATE, 14-1
  DROP, 14-2
  for stored PL/SQL units, 14-1
  in trigger bodies, 9-20, 9-25
  not allowed in triggers, 9-25
SQLCODE function
  links to examples, 13-125
  syntax, 13-125
SQLData Java storage format, 14-72
SQLERRM function
  links to examples, 13-127
  syntax, 13-126
  using with BULK_EXCEPTIONS ERROR_CODE field, 12-16
SQLJ object types
  creating, 14-72
  mapping a Java class to, 14-73
standalone procedures
  dropping, 14-93
STANDARD package
  defining PL/SQL environment, 10-9
START WITH clause, 6-5
STARTUP event
subprograms
  actual and formal parameters, 8-6
  advantages in PL/SQL, 8-2
  controlling side effects, 8-25
  declaring PL/SQL, 1-18
  default parameter modes, 8-9
  forward declaration of, 8-5
  guidelines for overloading, 8-13
  how calls are resolved, 8-16
  IN OUT parameter mode, 8-9
  IN parameter mode, 8-8
  in PL/SQL, 8-1
  invoking external, 8-23
  invoking from SQL*Plus, 1-20
  invoking with parameters, 8-11
  named parameters, 8-11
  OUT parameter mode, 8-8
  overloading names, 8-12
  parameter aliasing, 8-25
  parameter modes, 8-7, 8-9
  passing parameter by value, 8-25
  passing parameters by reference, 8-25
  positional parameters, 8-11
  recursive, 8-23
  restrictions on overloading, 8-14
  using database links with invoker’s-rights, 8-21
  using recursion, 8-23
  using triggers with invoker’s-rights, 8-20
  using views with invoker’s-rights, 8-20
subqueries
  correlated, 6-20
  using in PL/SQL, 6-19
SUBSCRIPT_BEYOND_COUNT exception, 11-5
SUBSCRIPT_OUTSIDE_LIMIT exception, 11-5
subtypes, 14-21
  CHARACTER, 3-9
  compatibility, 3-25
  constrained and unconstrained, 3-24
  defining, 3-24
  dropping safely, 14-96
  STRING, 3-9
  using, 3-24
  VARCHAR, 3-9
supertypes, 14-21
syntax
  BEGIN, 13-13
  collection method, 13-27
  exception definition, 13-44, 13-45
  FETCH statement, 13-68
  literal declaration, 13-85
  LOOP statement, 13-88
  NULL statement, 13-93
  reading diagrams, 13-1, 14-1
  WHILE-LOOP statement, 13-91
syntax of PL/SQL language elements, 13-1
SYS_INVALID_ROWID exception, 11-5
SYS_REFCURSOR type, 12-38
system events
  triggers on, 14-60
T
  table
  mutating, 9-25
  TABLE data type, 5-7
  table functions
  creating, 14-34
  exception handling, 12-42
  fetching from results of, 12-38
  for querying, 12-34
  organizing multiple calls to, 12-38
  passing data with cursor variables, 12-38
  performing DML operations inside, 12-41
  performing DML operations on, 12-41
  pipelining data between, 12-37
  returning results from, 12-37
  setting up transformation pipelines, 12-34
  using transformation pipelines, 12-36
  writing transformation pipelines, 12-35
  tables
  constraining, 9-25
  hash
    simulating with associative arrays, 5-5
    See associative arrays
  mutating, 9-25
  nested, 5-4
    characteristics of, 5-2
    creating, 14-78
    store, 5-6
  tabs, 2-2
  terminators, 2-3
  THEN clause
    using, 4-2
    with IF statement, 4-2
  TIMEOUT_ON_RESOURCE exception, 11-5
  TIMESTAMP data type, 3-17
  TIMESTAMP WITH LOCAL TIME ZONE data type, 3-19
  TIMESTAMP WITH TIME ZONE data type, 3-18
  TIMEZONE_ABBR data type field, 3-16
  TIMEZONE_HOUR data type field, 3-16
  TIMEZONE_MINUTES data type field, 3-16
  TIMEZONE_REGION data type field, 3-16
  TO_NUMBER function, 7-6
  TOO_MANY_ROWS exception, 11-6
  Trace API
    tracing code for tuning, 12-9
  tracking database events, 9-44
  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
triggers and, 9-2
visibility, 6-43
trigger
disabled
definition, 9-2
enabled
definition, 9-2
trigger body
defining, 14-62
triggering statement
definition, 9-6
Triggers
compound, 9-13
triggers
accessing column values, 9-20
AFTER, 9-7, 9-20, 9-32, 9-34, 14-56
as a stored PL/SQL subprogram, 1-20
auditing with, 9-31, 9-32
autonomous, 6-45
BEFORE, 9-7, 9-20, 9-41, 9-42, 14-55
body, 9-18, 9-22, 9-25
check constraints, 9-40, 9-41
column list in UPDATE, 9-7, 9-22
compiled, 9-27
compiling, 14-12
compound, 14-58
conditional predicates, 9-18, 9-22
constraints and, 9-3, 9-35
creating, 9-4, 9-5, 9-24, 14-51
creating enabled or disabled, 14-61
data access and, 9-2
data access restrictions, 9-41
database
altering, 14-13
dropping, 14-94
debugging, 9-29
designing, 9-3
disabling, 9-2, 14-12, 14-61
enabling, 9-2, 14-12, 14-13, 14-51
error conditions and exceptions, 9-22
events, 9-6
examples, 9-31 to 9-42
executing
with a PL/SQL block, 14-62
following other triggers, 14-61
FOR EACH ROW clause, 9-12
generating derived column values, 9-42
illegal SQL statements, 9-25
INSTEAD OF, 14-56
INSTEAD OF triggers, 9-8
listing information about, 9-30
modifying, 9-29
mutating tables and, 9-25
naming, 9-6
on database events, 14-60
on DML operations, 14-53, 14-57
on views, 14-56
package variables and, 9-7
privileges
to drop, 9-29
procedures and, 9-25
recompiling, 9-28
re-creating, 14-55
REFERENCING option, 9-21
referential integrity and, 9-36 to 9-39
remote dependencies and, 9-25
remote exceptions, 9-23
removing from the database, 14-94
renaming, 14-13
restrictions, 9-13, 9-24
restrictions on, 14-61
row, 9-12, 14-58
row evaluation order, 9-8
row values
old and new, 14-58
sequential, 14-61
SQL examples, 14-62
statement, 14-58
stored, 9-27
use of LONG and LONG RAW data types, 9-25
username reported in, 9-27
WHEN clause, 9-13
triggers on object tables, 9-22
TRIM method
collections, 5-26, 13-29
TRUE value, 2-8
TRUST attribute
of PRAGMA RESTRICT_REFERENCES, 14-76
TRUST option
RESTRICT_REFERENCES, 13-108
tuning
allocate large VARCHAR2 variables, 12-7
avoid memory overhead, 12-7
computation-intensive programs, 12-27
do not duplicate built-in functions, 12-5
dynamic SQL programs, 12-27
group related subprograms into a package, 12-7
guidelines for avoiding PL/SQL performance
problems, 12-3
improve code to avoid compiler warnings, 12-7
make function calls efficient, 12-4
make loops efficient, 12-5
make SQL statements efficient, 12-3
optimizing PL/SQL programs, 12-1
pin packages in the shared memory pool, 12-7
PL/SQL code, 12-2
profiling and tracing, 12-8
reducing loop overhead, 12-9
reorder conditional tests to put least expensive
first, 12-5
use BINARY_FLOAT or BINARY_DOUBLE for
floating-point arithmetic, 12-6
use PLS_INTEGER for integer arithmetic, 12-6
using DBMS_PROFILE and DBMS_TRACE, 12-8
using FORALL, 12-10
using NOCOPY, 12-28
using transformation pipelines, 12-34

TYPE attribute, 2-12
declaring, 1-10
links to examples, 13-129
syntax, 13-128
with SUBTYPE, 3-24

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
type methods
return type of, 14-74
types
composite, 5-1
types. See object types or data types

U
unbounded collections, 5-2
unhandled exceptions
catching, 11-16
propagating, 11-10
UNION ALL set operator, 6-7
UNION set operator, 6-7
universal rowids, 3-15
updatable view
definition, 9-8
UPDATE statement
column values and triggers, 9-20
triggers and, 9-7, 9-22
triggers for referential integrity, 9-37, 9-38
triggers on, 14-57
with a record variable, 5-37
URL (uniform resource locator), 10-11
UROWID
pseudocolumn, 6-5
UROWID data type, 3-14
user-defined
exceptions in PL/SQL, 11-6
user-defined aggregate functions, 14-35
user-defined statistics
dropping, 14-91, 14-95
user-defined types
defining, 14-71
mapping to Java classes, 14-72
usernames
as reported in a trigger, 9-27
USING clause
EXECUTE IMMEDIATE, 13-49
with OPEN FOR statement, 13-97
UTL_FILE package, 10-11
UTL_HTTP package, 10-11
UTL_SMTP package, 10-12
utlrp.sql script
for PL/SQL native compilation, 12-33

V
V$RESERVED_WORDS view, D-1
VALIDATE clause
drop TYPE, 14-96
validation checks
avoiding SQL injection with, 7-15
VALUE_ERROR exception, 11-6
VALUES OF clause, 12-10
FORALL, 13-72
VARCHAR subtype, 3-9
VARCHAR2 data type
differences with CHAR, 3-9
variables
assigning query result to, 2-27
assigning values, 1-7, 2-26
bind
See bind variables
composite, 5-1
declaring, 1-7, 2-10
global, 10-9
initializing, 2-26
links to examples, 13-34, 13-132
passing as IN OUT parameter, 1-8
REF CURSOR data type, 6-22
syntax, 13-32, 13-130
understanding PL/SQL, 1-6
variable-size arrays (varrays)
characteristics of, 5-2
VARRAY data type, 5-7
varrays
compared to nested tables, 5-6
creating, 14-65, 14-77
dropping the body of, 14-97
dropping the specification of, 14-95
increasing size of, 14-25
See variable-size arrays
size limit, 5-7
syntax, 13-22
TYPE definition, 5-7
views
containing expressions, 9-9
inherently modifiable, 9-9
modifiable, 9-9
pseudocolumns, 9-9
visibility
of package contents, 10-1
transaction, 6-43
visibility of identifier, 2-22

W
warning messages
controlling PL/SQL, 11-20
WHEN clause, 9-13
cannot contain PL/SQL expressions, 9-13
correlation names, 9-20
examples, 9-36
EXCEPTION examples, 9-23, 9-36, 9-40, 9-41
type handling, 13-45
type exceptions, 11-13
of CREATE TRIGGER, 14-61
using, 4-10, 4-11
WHILE-LOOP statement
  overview, 1-15
  syntax, 13-91
  using, 4-13
wildcards, 2-36
WNDS attribute
  of PRAGMA RESTRICT_REFERENCES, 14-76
WNDS option
  RESTRICT_REFERENCES, 13-108
WNPS attribute
  of PRAGMA RESTRICT_REFERENCES, 14-76
WNPS option
  RESTRICT_REFERENCES, 13-108
work areas
  queries, 6-23
wrap utility
  running, A-3
wrapping PL/SQL source code, A-1

Y

YEAR
  data type field, 3-16

Z

ZERO_DIVIDE exception, 11-6
ZONE
  part of TIMESTAMP data type, 3-18