PL/SQL

User’s Guide and Reference

Release 9.0.1

June 2001
Part No. A89856-01
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Us Your Comments .................................................................................................................. xix</td>
</tr>
<tr>
<td>Preface ........................................................................................................................................... xxi</td>
</tr>
<tr>
<td>What’s New in PL/SQL? .................................................................................................................... xxxi</td>
</tr>
<tr>
<td>1 Overview of PL/SQL</td>
</tr>
<tr>
<td>Understanding the Main Features of PL/SQL .................................................................................. 1-2</td>
</tr>
<tr>
<td>Block Structure ............................................................................................................................. 1-2</td>
</tr>
<tr>
<td>Variables and Constants .................................................................................................................. 1-3</td>
</tr>
<tr>
<td>Cursors ........................................................................................................................................ 1-5</td>
</tr>
<tr>
<td>Cursor FOR Loops ........................................................................................................................... 1-6</td>
</tr>
<tr>
<td>Cursor Variables ............................................................................................................................ 1-6</td>
</tr>
<tr>
<td>Attributes ..................................................................................................................................... 1-7</td>
</tr>
<tr>
<td>Control Structures ......................................................................................................................... 1-9</td>
</tr>
<tr>
<td>Modularity ..................................................................................................................................... 1-12</td>
</tr>
<tr>
<td>Data Abstraction ........................................................................................................................... 1-14</td>
</tr>
<tr>
<td>Information Hiding ....................................................................................................................... 1-16</td>
</tr>
<tr>
<td>Error Handling ............................................................................................................................... 1-16</td>
</tr>
<tr>
<td>PL/SQL Architecture ...................................................................................................................... 1-17</td>
</tr>
<tr>
<td>In the Oracle Database Server ...................................................................................................... 1-18</td>
</tr>
<tr>
<td>In Oracle Tools .............................................................................................................................. 1-20</td>
</tr>
<tr>
<td>Advantages of PL/SQL .................................................................................................................... 1-20</td>
</tr>
<tr>
<td>Support for SQL ............................................................................................................................. 1-20</td>
</tr>
</tbody>
</table>
Support for Object-Oriented Programming .......................................................... 1-21
Better Performance .................................................................................................. 1-21
Higher Productivity .................................................................................................. 1-22
Full Portability .......................................................................................................... 1-23
Tight Integration with SQL ....................................................................................... 1-23
Tight Security ............................................................................................................ 1-23

2 Fundamentals of PL/SQL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Set</td>
<td>2-2</td>
</tr>
<tr>
<td>Lexical Units</td>
<td>2-2</td>
</tr>
<tr>
<td>Delimiters</td>
<td>2-3</td>
</tr>
<tr>
<td>Identifiers</td>
<td>2-4</td>
</tr>
<tr>
<td>Literals</td>
<td>2-7</td>
</tr>
<tr>
<td>Comments</td>
<td>2-10</td>
</tr>
<tr>
<td>Declarations</td>
<td>2-11</td>
</tr>
<tr>
<td>Using DEFAULT</td>
<td>2-12</td>
</tr>
<tr>
<td>Using NOT NULL</td>
<td>2-13</td>
</tr>
<tr>
<td>Using %TYPE</td>
<td>2-13</td>
</tr>
<tr>
<td>Using %ROWTYPE</td>
<td>2-14</td>
</tr>
<tr>
<td>Restrictions on Declarations</td>
<td>2-17</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>2-17</td>
</tr>
<tr>
<td>Synonyms</td>
<td>2-18</td>
</tr>
<tr>
<td>Scoping</td>
<td>2-18</td>
</tr>
<tr>
<td>Case Sensitivity</td>
<td>2-18</td>
</tr>
<tr>
<td>Name Resolution</td>
<td>2-18</td>
</tr>
<tr>
<td>Scope and Visibility</td>
<td>2-19</td>
</tr>
<tr>
<td>Assignments</td>
<td>2-22</td>
</tr>
<tr>
<td>Boolean Values</td>
<td>2-22</td>
</tr>
<tr>
<td>Database Values</td>
<td>2-23</td>
</tr>
<tr>
<td>Expressions and Comparisons</td>
<td>2-23</td>
</tr>
<tr>
<td>Operator Precedence</td>
<td>2-24</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>2-25</td>
</tr>
<tr>
<td>Comparison Operators</td>
<td>2-26</td>
</tr>
<tr>
<td>Concatenation Operator</td>
<td>2-28</td>
</tr>
<tr>
<td>Boolean Expressions</td>
<td>2-28</td>
</tr>
</tbody>
</table>
3 PL/SQL Datatypes

Predefined Datatypes ................................................................. 3-2
  Number Types ........................................................................ 3-3
  Character Types ..................................................................... 3-5
  National Character Types ..................................................... 3-10
  LOB Types ............................................................................ 3-13
  Boolean Type .......................................................................... 3-15
  Datetime and Interval Types .................................................. 3-15
  Datetime and Interval Arithmetic ........................................... 3-20
  Avoiding Truncation Problems Using Date and Time Subtypes . 3-21

User-Defined Subtypes .............................................................. 3-21
  Defining Subtypes ............................................................... 3-22
  Using Subtypes ..................................................................... 3-22

Datatype Conversion ............................................................... 3-24
  Explicit Conversion ............................................................ 3-24
  Implicit Conversion ............................................................. 3-24
  Implicit versus Explicit Conversion ....................................... 3-26
  DATE Values .......................................................................... 3-26
  RAW and LONG RAW Values ................................................ 3-27

4 PL/SQL Control Structures

Overview of PL/SQL Control Structures ..................................... 4-2

Conditional Control: IF and CASE Statements ....................... 4-2
  IF-THEN Statement ............................................................. 4-3
  IF-THEN-ELSE Statement .................................................... 4-3
  IF-THEN-ELSEIF Statement .................................................. 4-4
  CASE Statement ................................................................. 4-5
  Guidelines for PL/SQL Conditional Statements ....................... 4-8

Iterative Control: LOOP and EXIT Statements ....................... 4-9
  LOOP .................................................................................. 4-9
  WHILE-LOOP ....................................................................... 4-11
FOR-LOOP ................................................................................................................................... 4-13
Sequential Control: GOTO and NULL Statements ................................................................. 4-17
GOTO Statement ...................................................................................................................... 4-17
NULL Statement ....................................................................................................................... 4-21

5 PL/SQL Collections and Records
What Is a Collection? ........................................................................................................... 5-2
Understanding Nested Tables ............................................................................................. 5-2
Nested Tables Versus Index-By Tables ................................................................................ 5-3
Understanding Varrays ......................................................................................................... 5-4
Varrays Versus Nested Tables ............................................................................................. 5-4
Defining and Declaring Collections .................................................................................... 5-5
Declaring Collections ............................................................................................................ 5-7
Initializing and Referencing Collections ........................................................................... 5-8
Referencing Collection Elements ...................................................................................... 5-10
Assigning and Comparing Collections .............................................................................. 5-11
Comparing Whole Collections ............................................................................................ 5-13
Manipulating Collections ................................................................................................... 5-13
Some Nested Table Examples ............................................................................................ 5-13
Some Varray Examples .......................................................................................................... 5-16
Manipulating Individual Collection Elements .................................................................... 5-18
Manipulating Local Collections .......................................................................................... 5-20
Using Multilevel Collections ............................................................................................. 5-21
Using Collection Methods ................................................................................................ 5-23
Using the EXISTS Collection Method ............................................................................... 5-24
Using the COUNT Collection Method ................................................................................ 5-24
Using the LIMIT Collection Method ................................................................................ 5-25
Using the FIRST and LAST Collection Methods ............................................................... 5-25
Using the PRIOR and NEXT Collection Methods .............................................................. 5-26
Using the EXTEND Collection Method ............................................................................ 5-27
Using the TRIM Collection Method .................................................................................... 5-27
Using the DELETE Collection Method .............................................................................. 5-28
Applying Methods to Collection Parameters ..................................................................... 5-29
Avoiding Collection Exceptions .......................................................................................... 5-30
Reducing Loop Overhead for Collections with Bulk Binds ................................................ 5-32
How Do Bulk Binds Improve Performance? ................................................................. 5-33

Using the FORALL Statement ..................................................................................... 5-34
  How FORALL Affects Rollbacks ........................................................................... 5-35
  Counting Rows Affected by FORALL Iterations with the %BULK_ROWCOUNT Attribute ... 5-36
  Handling FORALL Exceptions with the %BULK.Exceptions Attribute ................. 5-38
  Restrictions on FORALL ..................................................................................... 5-39

Retrieving Query Results into Collections with the BULK COLLECT Clause .......... 5-41
  Example of a Bulk Fetch from a Cursor ............................................................... 5-42
  Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause .......... 5-43
  Retrieving DML Results into a Collection with the RETURNING INTO Clause ...... 5-43
  Restrictions on BULK COLLECT ....................................................................... 5-44
  Using FORALL and BULK COLLECT Together ............................................... 5-45
  Using Host Arrays with Bulk Binds ................................................................. 5-45

What Is a Record? ..................................................................................................... 5-46

Defining and Declaring Records ........................................................................... 5-46
  Declaring Records ............................................................................................. 5-48
  Initializing Records .......................................................................................... 5-48
  Referencing Records ........................................................................................ 5-49
  Nulling Records ............................................................................................... 5-51
  Assigning Records ............................................................................................ 5-51
  Comparing Records .......................................................................................... 5-53

Manipulating Records ............................................................................................ 5-53

6 Interaction Between PL/SQL and Oracle

Overview of SQL Support in PL/SQL ................................................................. 6-2
  Data Manipulation ............................................................................................ 6-2
  Transaction Control ......................................................................................... 6-2
  SQL Functions ................................................................................................. 6-2
  SQL Pseudocolumns ....................................................................................... 6-3
  SQL Operators ................................................................................................. 6-5

Managing Cursors ................................................................................................ 6-6
  Overview of Explicit Cursors ......................................................................... 6-6
  Overview of Implicit Cursors ......................................................................... 6-11

Separating Cursor Specs and Bodies with Packages ........................................... 6-12
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some Examples of Package Features</td>
<td>9-9</td>
</tr>
<tr>
<td>Private Versus Public Items in Packages</td>
<td>9-14</td>
</tr>
<tr>
<td>Overloading Packaged Subprograms</td>
<td>9-15</td>
</tr>
<tr>
<td>How Package STANDARD Defines the PL/SQL Environment</td>
<td>9-15</td>
</tr>
<tr>
<td>Overview of Product-Specific Packages</td>
<td>9-16</td>
</tr>
<tr>
<td>About the DBMS_ALERT Package</td>
<td>9-16</td>
</tr>
<tr>
<td>About the DBMS_OUTPUT Package</td>
<td>9-16</td>
</tr>
<tr>
<td>About the DBMS_PIPE Package</td>
<td>9-17</td>
</tr>
<tr>
<td>About the UTL_FILE Package</td>
<td>9-17</td>
</tr>
<tr>
<td>About the UTL_HTTP Package</td>
<td>9-18</td>
</tr>
<tr>
<td>Guidelines for Writing Packages</td>
<td>9-18</td>
</tr>
</tbody>
</table>

**10 PL/SQL Object Types**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Role of Abstraction</td>
<td>10-2</td>
</tr>
<tr>
<td>What Is an Object Type?</td>
<td>10-3</td>
</tr>
<tr>
<td>Why Use Object Types?</td>
<td>10-5</td>
</tr>
<tr>
<td>Structure of an Object Type</td>
<td>10-5</td>
</tr>
<tr>
<td>Components of an Object Type</td>
<td>10-7</td>
</tr>
<tr>
<td>Attributes</td>
<td>10-7</td>
</tr>
<tr>
<td>Methods</td>
<td>10-8</td>
</tr>
<tr>
<td>Changing Attributes and Methods of an Existing Object Type (Type Evolution)</td>
<td>10-12</td>
</tr>
<tr>
<td>Defining Object Types</td>
<td>10-13</td>
</tr>
<tr>
<td>Overview of PL/SQL Type Inheritance</td>
<td>10-14</td>
</tr>
<tr>
<td>Object Type Example: Stack</td>
<td>10-16</td>
</tr>
<tr>
<td>Object Type Example: Ticket_Booth</td>
<td>10-18</td>
</tr>
<tr>
<td>Object Type Example: Bank_Account</td>
<td>10-20</td>
</tr>
<tr>
<td>Object Type Example: Rational Numbers</td>
<td>10-22</td>
</tr>
<tr>
<td>Declaring and Initializing Objects</td>
<td>10-24</td>
</tr>
<tr>
<td>Declaring Objects</td>
<td>10-25</td>
</tr>
<tr>
<td>Initializing Objects</td>
<td>10-25</td>
</tr>
<tr>
<td>How PL/SQL Treats Uninitialized Objects</td>
<td>10-26</td>
</tr>
<tr>
<td>Accessing Attributes</td>
<td>10-27</td>
</tr>
<tr>
<td>Calling Constructors</td>
<td>10-28</td>
</tr>
<tr>
<td>Calling Methods</td>
<td>10-28</td>
</tr>
<tr>
<td>Sharing Objects</td>
<td>10-29</td>
</tr>
</tbody>
</table>
Using Refs ................................................................. 10-31
Forward Type Definitions ........................................... 10-31
Manipulating Objects .................................................... 10-32
Selecting Objects .......................................................... 10-33
Inserting Objects ............................................................ 10-38
Updating Objects ........................................................... 10-39
Deleting Objects ............................................................. 10-39

11 Native Dynamic SQL

What Is Dynamic SQL? .................................................. 11-2
The Need for Dynamic SQL ........................................... 11-2
Using the EXECUTE IMMEDIATE Statement ..................... 11-3
Some Examples of Dynamic SQL ...................................... 11-5
Backward Compatibility of the USING Clause .................... 11-6
Specifying Parameter Modes ........................................... 11-7
Using the OPEN-FOR, FETCH, and CLOSE Statements ........... 11-8
Opening the Cursor Variable .......................................... 11-8
Fetching from the Cursor Variable ................................... 11-9
Closing the Cursor Variable .......................................... 11-9
Examples of Dynamic SQL for Records, Objects, and Collections .... 11-10
Using Bulk Dynamic SQL .............................................. 11-12
Syntax for Dynamic Bulk Binds ........................................ 11-12
Examples of Dynamic Bulk Binds ..................................... 11-13
Tips and Traps for Dynamic SQL ..................................... 11-15
Improving Performance ............................................... 11-15
Making Procedures Work on Arbitrarily Named Schema Objects .... 11-15
Using Duplicate Placeholders ........................................ 11-16
Using Cursor Attributes .............................................. 11-17
Passing Nulls ............................................................. 11-17
Doing Remote Operations ............................................ 11-18
Using Invoker Rights ................................................ 11-18
Using Pragma RESTRICT_REFERENCES ....................... 11-19
Avoiding Deadlocks .................................................... 11-19
12 Tuning PL/SQL Applications

Reasons for PL/SQL Performance Problems ................................................................. 12-2
Identifying PL/SQL Performance Problems ................................................................. 12-8
  The Profiler API: Package DBMS_PROFILER ......................................................... 12-8
  The Trace API: Package DBMS_TRACE ...................................................................... 12-9
PL/SQL Features for Performance Tuning ................................................................. 12-10
  Improving PL/SQL Performance with Native Dynamic SQL ............................... 12-10
  Improving PL/SQL Performance with Bulk Binds ................................................... 12-11
  Improving PL/SQL Performance with the NOCOPY Compiler Hint ....................... 12-12
  Improving PL/SQL Performance with the RETURNING Clause ......................... 12-12
  Improving PL/SQL Performance with External Routines ....................................... 12-13
  Improving PL/SQL Performance with Object Types and Collections ............... 12-13
  Compiling PL/SQL Code for Native Execution ...................................................... 12-14

13 PL/SQL Language Elements

Assignment Statement .................................................................................................. 13-4
AUTONOMOUS_TRANSACTION Pragma ................................................................. 13-8
Blocks ....................................................................................................................... 13-11
CASE Statement ....................................................................................................... 13-18
CLOSE Statement ...................................................................................................... 13-21
Collection Methods ................................................................................................ 13-23
Collections ............................................................................................................... 13-28
Comments ................................................................................................................ 13-35
COMMIT Statement .................................................................................................. 13-36
Constants and Variables .......................................................................................... 13-38
Cursor Attributes ...................................................................................................... 13-42
Cursor Variables ...................................................................................................... 13-47
Cursors ...................................................................................................................... 13-53
DELETE Statement ................................................................................................. 13-57
EXCEPTION_INIT Pragma ......................................................................................... 13-61
Exceptions ............................................................................................................... 13-63
EXECUTE IMMEDIATE Statement ........................................................................... 13-66
EXIT Statement ........................................................................................................ 13-70
Expressions ............................................................................................................. 13-72
FETCH Statement ..................................................................................................... 13-83
# FORALL Statement

FORALL Statement ................................................................. 13-88

# Functions

Functions .................................................................................. 13-90

# GOTO Statement

GOTO Statement ........................................................................ 13-97

# IF Statement

IF Statement ............................................................................... 13-99

# INSERT Statement

INSERT Statement ...................................................................... 13-102

# Literals

Literals ....................................................................................... 13-105

# LOCK TABLE Statement

LOCK TABLE Statement .......................................................... 13-108

# LOOP Statements

LOOP Statements ....................................................................... 13-110

# MERGE Statement

MERGE Statement ...................................................................... 13-117

# NULL Statement

NULL Statement ......................................................................... 13-119

# Object Types

Object Types .............................................................................. 13-120

# OPEN Statement

OPEN Statement .......................................................................... 13-129

# OPEN-FOR Statement

OPEN-FOR Statement ............................................................. 13-131

# OPEN-FOR-USING Statement

OPEN-FOR-USING Statement .................................................. 13-134

# Packages

Packages ..................................................................................... 13-137

# Procedures

Procedures ................................................................................ 13-143

# RAISE Statement

RAISE Statement ....................................................................... 13-148

# Records

Records ..................................................................................... 13-150

# RESTRICT_REFERENCES Pragma

RESTRICT_REFERENCES Pragma .............................................. 13-154

# RETURN Statement

RETURN Statement ..................................................................... 13-157

# ROLLBACK Statement

ROLLBACK Statement .................................................................. 13-159

# %ROWTYPE Attribute

%ROWTYPE Attribute .................................................................. 13-161

# SAVEPOINT Statement

SAVEPOINT Statement ................................................................ 13-163

# SELECT INTO Statement

SELECT INTO Statement .......................................................... 13-164

# SERIALLY_REUSABLE Pragma

SERIALLY_REUSABLE Pragma ................................................ 13-169

# SET TRANSACTION Statement

SET TRANSACTION Statement .................................................. 13-171

# SQL Cursor

SQL Cursor ............................................................................... 13-173

# SQLCODE Function

SQLCODE Function ..................................................................... 13-176

# SQLERRM Function

SQLERRM Function .................................................................... 13-178

# %TYPE Attribute

%TYPE Attribute ........................................................................ 13-180

# UPDATE Statement

UPDATE Statement ...................................................................... 13-182

---

**A Sample PL/SQL Programs**

Running the Programs ............................................................. A-2

Sample 1. FOR Loop ................................................................. A-3
D  PL/SQL Name Resolution

What Is Name Resolution?............................................................................................................ D-2
Various Forms of References...................................................................................................... D-3
Name-Resolution Algorithm ...................................................................................................... D-4
  Finding the Basis....................................................................................................................... D-5
Understanding Capture ........................................................................................................... D-7
  Inner Capture ......................................................................................................................... D-7
  Same-Scope Capture ............................................................................................................... D-8
  Outer Capture ........................................................................................................................... D-9
Avoiding Capture ....................................................................................................................... D-9
Accessing Attributes and Methods .......................................................................................... D-9
Calling Subprograms and Methods ......................................................................................... D-10
Name Resolution for SQL Versus PL/SQL .............................................................................. D-12

E  PL/SQL Program Limits

F  PL/SQL Reserved Words

Index
Send Us Your Comments

PL/SQL User’s Guide and Reference, Release 9.0.1
Part No. A89856-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

■ Did you find any errors?
■ Is the information clearly presented?
■ Do you need more information? If so, where?
■ Are the examples correct? Do you need more examples?
■ What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

■ Electronic mail: infodev_us@oracle.com
■ FAX: (650) 506-7227   Attn: Server Technologies Documentation Manager
■ Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA  94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.
Preface

PL/SQL, Oracle’s procedural extension of SQL, is an advanced fourth-generation programming language (4GL). It offers modern features such as data encapsulation, overloading, collection types, exception handling, and information hiding. PL/SQL also offers seamless SQL access, tight integration with the Oracle server and tools, portability, and security.

This guide explains all the concepts behind PL/SQL and illustrates every facet of the language. Good programming style is stressed throughout and supported by numerous examples. Using this guide, you learn PL/SQL quickly and efficiently.

This preface contains these topics:

- Audience
- Organization
- Related Documentation
- Conventions
- Sample Database Tables
- Documentation Accessibility
**Audience**

Anyone developing PL/SQL-based applications for Oracle will benefit from reading this guide. Written especially for programmers, this comprehensive treatment of PL/SQL will also be of value to systems analysts, project managers, and others interested in database applications. To use this guide effectively, you need a working knowledge of Oracle, SQL, and a 3GL such as Ada, C, or COBOL.

You will not find installation instructions or system-specific information in this guide. For that kind of information, see the Oracle installation or user’s guide for your system.

**Organization**

The *PL/SQL User’s Guide and Reference* contains:

**Chapter 1, "Overview of PL/SQL"**

This chapter surveys the main features of PL/SQL and points out the advantages they offer. It also acquaints you with the basic concepts behind PL/SQL and the general appearance of PL/SQL programs.

**Chapter 2, "Fundamentals of PL/SQL"**

This chapter focuses on the small-scale aspects of PL/SQL. It discusses lexical units, scalar datatypes, user-defined subtypes, data conversion, expressions, assignments, block structure, declarations, and scope.

**Chapter 3, "PL/SQL Datatypes"**

This chapter discusses PL/SQL’s predefined datatypes, which include integer, floating point, character, Boolean, date, collection, reference, and LOB types. It also discusses user-defined subtypes and data conversion.

**Chapter 4, "PL/SQL Control Structures"**

This chapter shows you how to structure the flow of control through a PL/SQL program. It describes conditional, iterative, and sequential control. You learn how to apply simple but powerful control structures such as *IF-THEN-ELSE*, *CASE*, and *WHILE-LOOP*.

**Chapter 5, "PL/SQL Collections and Records"**

This chapter focuses on the composite datatypes *TABLE*, *VARRAY*, and *RECORD*. You learn how to reference and manipulate whole collections of data, and how to treat related but dissimilar data as a logical unit. You also learn how to improve performance by bulk-binding collections.
Chapter 6, "Interaction Between PL/SQL and Oracle"
This chapter shows you how PL/SQL supports the SQL commands, functions, and operators that let you manipulate Oracle data. You also learn how to manage cursors, process transactions, and safeguard your database.

Chapter 7, "Handling PL/SQL Errors"
This chapter provides an in-depth discussion of error reporting and recovery. You learn how to detect and handle errors using PL/SQL exceptions.

Chapter 8, "PL/SQL Subprograms"
This chapter shows you how to write and use subprograms. It discusses procedures, functions, forward declarations, actual and formal parameters, positional and named notation, parameter modes, the NOCOPY compiler hint, parameter default values, aliasing, overloading, invoker rights, and recursion.

Chapter 9, "PL/SQL Packages"
This chapter shows you how to bundle related PL/SQL types, items, and subprograms into a package. Once written, your general-purpose package is compiled, then stored in an Oracle database, where its contents can be shared by many applications.

Chapter 10, "PL/SQL Object Types"
This chapter introduces you to object-oriented programming based on object types, which provide abstract templates for real-world objects. You learn how to define object types and manipulate objects.

Chapter 11, "Native Dynamic SQL"
This chapter shows you how to use dynamic SQL, an advanced programming technique that makes your applications more flexible and versatile. You learn two simple ways to write programs that can build and process SQL statements "on the fly" at run time.

Chapter 12, "Tuning PL/SQL Applications"
This chapter shows you how to tune PL/SQL-based applications. You learn how to make small adjustments that improve performance.

Chapter 13, "PL/SQL Language Elements"
This chapter uses syntax diagrams to show how commands, parameters, and other language elements are sequenced to form PL/SQL statements. Also, it provides usage notes and short examples to help you become fluent in PL/SQL quickly.
Appendix A, "Sample PL/SQL Programs"
This appendix provides several PL/SQL programs to guide you in writing your own. The sample programs illustrate important concepts and features.

Appendix B, "CHAR versus VARCHAR2 Semantics"
This appendix explains the subtle but important semantic differences between the CHAR and VARCHAR2 base types.

Appendix C, "PL/SQL Wrap Utility"
This appendix shows you how to run the Wrap Utility, a standalone programming utility that enables you to deliver PL/SQL applications without exposing your source code.

Appendix D, "PL/SQL Name Resolution"
This appendix explains how PL/SQL resolves references to names in potentially ambiguous SQL and procedural statements.

Appendix E, "PL/SQL Program Limits"
This appendix helps you deal with the program limits imposed by the PL/SQL compilation and run-time system.

Appendix F, "PL/SQL Reserved Words"
This appendix lists those words reserved for use by PL/SQL.

Related Documentation
For more information, see these Oracle resources:

Various aspects of PL/SQL programming, in particular details for triggers and stored procedures, are covered in Oracle9i Application Developer's Guide - Fundamentals

For extensive information on object-oriented programming using both PL/SQL and SQL features, see Oracle9i Application Developer's Guide - Object-Relational Features

For information about programming with large objects (LOBs), see Oracle9i Application Developer's Guide - Large Objects (LOBs)

For SQL information, see the Oracle9i SQL Reference and Oracle9i Database Administrator's Guide. For basic Oracle concepts, see Oracle9i Database Concepts.

In North America, printed documentation is available for sale in the Oracle Store at
Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from
http://www.oraclebookshop.com/

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at
http://technet.oracle.com/membership/index.htm

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at
http://technet.oracle.com/docs/index.htm

or to the documentation search engine at:
http://tahiti.oracle.com/

This search engine has a number of features that you might find useful, such as searching for examples, looking up SQL and PL/SQL syntax, and formatting large numbers of search results into a "virtual book".

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.
<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.</td>
<td>When you specify this clause, you create an index-organized table.</td>
</tr>
<tr>
<td><em>Italics</em></td>
<td>Italic typeface indicates book titles or emphasis.</td>
<td>Oracle9i Database Concepts Ensure that the recovery catalog and target database do not reside on the same disk.</td>
</tr>
<tr>
<td><strong>UPPERCASE monospace (fixed-width font)</strong></td>
<td>Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.</td>
<td>You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_ TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.</td>
</tr>
<tr>
<td><strong>lowercase monospace (fixed-width font)</strong></td>
<td>Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. <strong>Note:</strong> Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.</td>
<td>Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the datafiles and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JRepUtil class implements these methods.</td>
</tr>
<tr>
<td><strong>lowercase monospace (fixed-width font) italic</strong></td>
<td>Lowercase monospace italic font represents placeholders or variables.</td>
<td>You can specify the parallel_clause. Run Uold_release.SQL where old_release refers to the release you installed prior to upgrading.</td>
</tr>
</tbody>
</table>

**Conventions in Code Examples**

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```sql
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```
The following table describes typographic conventions used in code examples and provides examples of their use.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Brackets enclose one or more optional items. Do not enter the brackets.</td>
<td>DECIMAL (digits [ , precision ])</td>
</tr>
<tr>
<td>{}</td>
<td>Braces enclose two or more items, one of which is required. Do not enter the braces.</td>
<td>{ENABLE</td>
</tr>
</tbody>
</table>
| |         | A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar. | {ENABLE | DISABLE}  
|            |                         | [COMPRESS | NOCOMPRESS] |
| ...        | Horizontal ellipsis points indicate either: | CREATE TABLE ... AS subquery; |
|            | ■ That we have omitted parts of the code that are not directly related to the example | SELECT col1, col2, ..., coln FROM employees; |
|            | ■ That you can repeat a portion of the code | |
| .          | Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example. | |
| Other notation | You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown. | acctbal NUMBER(11,2);  
|              | acct CONSTANT NUMBER(4) := 3; | |
| Italics    | Italicized text indicates placeholders or variables for which you must supply particular values. | CONNECT SYSTEM/system_password  
|            | DB_NAME = database_name | |
| UPPERCASE  | Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase. | SELECT last_name, employee_id FROM employees;  
|            | SELECT * FROM USER_TABLES;  
|            | DROP TABLE hr.employees; | |
Sample Database Tables

Most programming examples in this guide use two sample database tables named dept and emp. Their definitions follow:

```sql
CREATE TABLE dept (
    deptno NUMBER(2) NOT NULL,
    dname  VARCHAR2(14),
    loc    VARCHAR2(13));

CREATE TABLE emp (
    empno    NUMBER(4) NOT NULL,
    ename    VARCHAR2(10),
    job      VARCHAR2(9),
    mgr      NUMBER(4),
    hiredate DATE,
    sal      NUMBER(7,2),
    comm     NUMBER(7,2),
    deptno   NUMBER(2));
```

Sample Data

Respectively, the dept and emp tables contain the following rows of data:

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>EMPNO</td>
<td>ENAME</td>
<td>JOB</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
</tr>
</tbody>
</table>

**Documentation Accessibility**

Oracle’s goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. 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. 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 additional information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

JAWS, a Windows screen reader, 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, JAWS may not always read a line of text that consists solely of a bracket or brace.
This section describes new features of PL/SQL release 9.0.1 and provides pointers to additional information.

The following sections describe the new features in PL/SQL:

- **Oracle9i New Features in PL/SQL**
Oracle9i New Features in PL/SQL

■ Integration of SQL and PL/SQL parsers

   PL/SQL now supports the complete range of syntax for SQL statements, such as INSERT, UPDATE, DELETE, and so on. If you received errors for valid SQL syntax in PL/SQL programs before, those statements should now work.

   See Also: Because of more consistent error-checking, you might find that some invalid code is now found at compile time instead of producing an error at runtime, or vice versa. You might need to change the source code as part of the migration procedure. See Oracle9i Database Migration for details on the complete migration procedure.

■ CASE statements and expressions

   CASE statements and expressions are a shorthand way of representing IF/THEN choices with multiple alternatives.

   See Also:
   - "CASE Expressions" on page 2-31
   - "CASE Statement" on page 4-5
   - "CASE Statement" on page 13-18

■ Inheritance and Dynamic Method Dispatch

   Types can be declared in a supertype/subtype hierarchy, with subtypes inheriting attributes and methods from their supertypes. The subtypes can also add new attributes and methods, and override existing methods. A call to an object method executes the appropriate version of the method, based on the type of the object.

   See Also:
   - "Overview of PL/SQL Type Inheritance" on page 10-14
   - "How Overloading Works with Inheritance" on page 8-28

■ Type Evolution

   Attributes and methods can be added to and dropped from object types, without the need to re-create the types and corresponding data. This feature lets
the type hierarchy adapt to changes in the application, rather than being planned out entirely in advance.

See Also:  "Changing Attributes and Methods of an Existing Object Type (Type Evolution)" on page 10-12

■ New Date/Time Types

The new datatype `TIMESTAMP` records time values including fractional seconds. New datatypes `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` allow you to adjust date and time values to account for time zone differences. You can specify whether the time zone observes daylight savings time, to account for anomalies when clocks shift forward or backward. New datatypes `INTERVAL DAY TO SECOND` and `INTERVAL YEAR TO MONTH` represent differences between two date and time values, simplifying date arithmetic.

See Also:
- "Datetime and Interval Types" on page 3-15
- "Datetime and Interval Arithmetic" on page 3-20
- "Datetime Literals" on page 2-9

■ Native Compilation of PL/SQL Code

Improve performance by compiling Oracle-supplied and user-written stored procedures into native executables, using typical C development tools. This setting is saved so that the procedure is compiled the same way if it is later invalidated.

See Also:  "Compiling PL/SQL Code for Native Execution’ on page 12-14

■ Improved Globalization and National Language Support

Data can be stored in Unicode format using fixed-width or variable-width character sets. String handling and storage declarations can be specified using byte lengths, or character lengths where the number of bytes is computed for you. You can set up the entire database to use the same length semantics for strings, or specify the settings for individual procedures; this setting is remembered if a procedure is invalidated.
Table Functions and Cursor Expressions

You can query a set of returned rows like a table. Result sets can be passed from one function to another, letting you set up a sequence of transformations with no table to hold intermediate results. Rows of the result set can be returned a few at a time, reducing the memory overhead for producing large result sets within a function.

See Also:
- "Accepting and Returning Multiple Rows with Table Functions" on page 8-30
- "Using Cursor Expressions" on page 6-41

Multilevel Collections

You can nest the collection types, for example to create a VARRAY of PL/SQL tables, a VARRAY of VARRAYs, or a PL/SQL table of PL/SQL tables. You can model complex data structures such as multidimensional arrays in a natural way.

See Also:  "Using Multilevel Collections" on page 5-21

Better Integration for LOB Datatypes

You can operate on LOB types much like other similar types. You can use character functions on CLOB and NCLOB types. You can treat BLOB types as RAWs. Conversions between LOBs and other types are much simpler, particularly when converting from LONG to LOB types.

See Also:  "LOB Types" on page 3-13

Enhancements to Bulk Operations

You can now perform bulk SQL operations, such as bulk fetches, using native dynamic SQL (the EXECUTE IMMEDIATE statement). You can perform bulk insert or update operations that continue despite errors on some rows, then examine the problems after the operation is complete.
See Also:

- "Reducing Loop Overhead for Collections with Bulk Binds" on page 5-32
- "Using Bulk Dynamic SQL" on page 11-12
- "EXECUTE IMMEDIATE Statement" on page 13-66

- **MERGE Statement**

This specialized statement combines insert and update into a single operation. It is intended for data warehousing applications that perform particular patterns of inserts and updates.

See Also:

- "MERGE Statement" on page 13-117 for a brief discussion and example
- *Oracle9i SQL Reference* for detailed information
The limits of my language mean the limits of my world. —Ludwig Wittgenstein

This chapter surveys the main features of PL/SQL and points out the advantages they offer. It also acquaints you with the basic concepts behind PL/SQL and the general appearance of PL/SQL programs. You see how PL/SQL bridges the gap between database technology and procedural programming languages.

This chapter discusses the following topics:
- Understanding the Main Features of PL/SQL
- PL/SQL Architecture
- Advantages of PL/SQL

Understanding the Main Features of PL/SQL

A good way to get acquainted with PL/SQL is to look at a sample program. The program below processes an order for a tennis racket. First, it declares a variable of type NUMBER to store the quantity of tennis rackets on hand. Then, it retrieves the quantity on hand from a database table named inventory. If the quantity is greater than zero, the program updates the table and inserts a purchase record into another table named purchase_record. Otherwise, the program inserts an out-of-stock record into the purchase_record table.

```
-- available online in file 'examp1'
DECLARE
  qty_on_hand  NUMBER(5);
BEGIN
  SELECT quantity INTO qty_on_hand FROM inventory
  WHERE product = 'TENNIS RACKET'
  FOR UPDATE OF quantity;
  IF qty_on_hand > 0 THEN  -- check quantity
    UPDATE inventory SET quantity = quantity - 1
    WHERE product = 'TENNIS RACKET';
    INSERT INTO purchase_record
    VALUES ('Tennis racket purchased', SYSDATE);
  ELSE
    INSERT INTO purchase_record
    VALUES ('Out of tennis rackets', SYSDATE);
  END IF;
  COMMIT;
END;
```

With PL/SQL, you can use SQL statements to manipulate Oracle data and flow-of-control statements to process the data. Moreover, you can declare constants and variables, define procedures and functions, and trap runtime errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

Block Structure

PL/SQL is a block-structured language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or subproblem to be solved. Thus, PL/SQL supports the divide-and-conquer approach to problem solving called stepwise refinement.
A block (or sub-block) lets you group logically related declarations and statements. That way, you can place declarations close to where they are used. The declarations are local to the block and cease to exist when the block completes.

As Figure 1–1 shows, a PL/SQL block has three parts: a declarative part, an executable part, and an exception-handling part. (In PL/SQL, a warning or error condition is called an exception.) Only the executable part is required.

The order of the parts is logical. First comes the declarative part, in which items can be declared. Once declared, items can be manipulated in the executable part. Exceptions raised during execution can be dealt with in the exception-handling part.

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

Variables and Constants

PL/SQL lets you declare constants and variables, then use them in SQL and procedural statements anywhere an expression can be used. However, forward references are not allowed. So, you must declare a constant or variable before referencing it in other statements, including other declarative statements.
Declaring Variables

Variables can have any SQL datatype, such as CHAR, DATE, or NUMBER, or any PL/SQL datatype, such as BOOLEAN or BINARY_INTEGER. For example, assume that you want to declare a variable named part_no to hold 4-digit numbers and a variable named in_stock to hold the Boolean value TRUE or FALSE. You declare these variables as follows:

```
part_no  NUMBER(4);
in_stock BOOLEAN;
```

You can also declare nested tables, variable-size arrays (varrays for short), and records using the TABLE, VARRAY, and RECORD composite datatypes.

Assigning Values to a Variable

You can assign values to a variable in three ways. The first way uses the assignment operator (:=), a colon followed by an equal sign. You place the variable to the left of the operator and an expression (which can include function calls) to the right. A few examples follow:

```
tax := price * tax_rate;
valid_id := FALSE;
bonus := current_salary * 0.10;
wages := gross_pay(emp_id, st_hrs, ot_hrs) - deductions;
```

The second way to assign values to a variable is by selecting (or fetching) database values into it. In the example below, you have Oracle compute a 10% bonus when you select the salary of an employee. Now, you can use the variable bonus in another computation or insert its value into a database table.

```
SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id;
```

The third way to assign values to a variable is by passing it as an OUT or IN OUT parameter to a subprogram. As the following example shows, an IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller:

```
DECLARE
  my_sal REAL(7,2);
  my_salary (emp_id INT, salary IN OUT REAL) IS ...
BEGIN
  SELECT AVG(sal) INTO my_sal FROM emp;
  adjust_salary(7788, my_sal); -- assigns a new value to my_sal
```
Declaring Constants

Declaring a constant is like declaring a variable except that you must add the keyword CONSTANT and immediately assign a value to the constant. Thereafter, no more assignments to the constant are allowed. In the following example, you declare a constant named credit_limit:

```sql
credit_limit CONSTANT REAL := 5000.00;
```

Cursors

Oracle uses work areas to execute SQL statements and store processing information. A PL/SQL construct called a cursor lets you name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, 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. An example follows:

```sql
DECLARE
  CURSOR c1 IS
    SELECT empno, ename, job FROM emp WHERE deptno = 20;

The set of rows returned by a multi-row query is called the result set. Its size is the number of rows that meet your search criteria. As Figure 1–2 shows, an explicit cursor "points" to the current row in the result set. This allows your program to process the rows one at a time.

Figure 1–2 Query Processing

```

Multi-row query processing is somewhat like file processing. For example, a COBOL program opens a file, processes records, then closes the file. Likewise, a PL/SQL program opens a cursor, processes rows returned by a query, then closes the cursor. Just as a file pointer marks the current position in an open file, a cursor marks the current position in a result set.
You use the OPEN, FETCH, and CLOSE statements to control a cursor. The OPEN statement executes the query associated with the cursor, identifies the result set, and positions the cursor before the first row. The FETCH statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the CLOSE statement disables the cursor.

Cursor FOR Loops

In most situations that require an explicit cursor, you can simplify coding by using a cursor FOR loop instead of the OPEN, FETCH, and CLOSE statements. A cursor FOR loop implicitly declares its loop index as a record that represents a row fetched from the database. Next, it opens a cursor, repeatedly fetches rows of values from the result set into fields in the record, then closes the cursor when all rows have been processed. In the following example, the cursor FOR loop implicitly declares emp_rec as a record:

```
DECLARE
  CURSOR c1 IS
    SELECT ename, sal, hiredate, deptno FROM emp;
  ...
BEGIN
  FOR emp_rec IN c1 LOOP
    ...
    salary_total := salary_total + emp_rec.sal;
  END LOOP;
```

To reference individual fields in the record, you use dot notation, in which a dot (.) serves as the component selector.

Cursor Variables

Like a cursor, a cursor variable points to the current row in the result set of a multi-row query. But, unlike a cursor, a cursor variable can be opened for any type-compatible query. It is not tied to a specific query. Cursor variables are true PL/SQL variables, to which you can assign new values and which you can pass to subprograms stored in an Oracle database. This gives you more flexibility and a convenient way to centralize data retrieval.
Typically, you open a cursor variable by passing it to a stored procedure that declares a cursor variable as one of its formal parameters. The following procedure opens the cursor variable `generic_cv` for the chosen query:

```
PROCEDURE open_cv (generic_cv IN OUT GenericCurTyp, choice NUMBER) IS
BEGIN
  IF choice = 1 THEN
    OPEN generic_cv FOR SELECT * FROM emp;
  ELSIF choice = 2 THEN
    OPEN generic_cv FOR SELECT * FROM dept;
  ELSIF choice = 3 THEN
    OPEN generic_cv FOR SELECT * FROM salgrade;
  END IF;
... IF;
END;
```

**Attributes**

PL/SQL variables and cursors have attributes, which are properties that let you reference the datatype and structure of an item without repeating its definition. Database columns and tables have similar attributes, which you can use to ease maintenance. A percent sign (%) serves as the attribute indicator.

**%TYPE**

The `%TYPE` attribute provides the datatype of a variable or database column. This is particularly useful when declaring variables that will hold database values. For example, assume there is a column named `title` in a table named `books`. To declare a variable named `my_title` that has the same datatype as column `title`, use dot notation and the `%TYPE` attribute, as follows:

```
my_title books.title%TYPE;
```

Declaring `my_title` with `%TYPE` has two advantages. First, you need not know the exact datatype of `title`. Second, if you change the database definition of `title` (make it a longer character string for example), the datatype of `my_title` changes accordingly at run time.

**%ROWTYPE**

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The `%ROWTYPE` attribute provides a record type that represents a row in a table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable.
Columns in a row and corresponding fields in a record have the same names and datatypes. In the example below, you declare a record named dept_rec. Its fields have the same names and datatypes as the columns in the dept table.

```plsql
DECLARE
    dept_rec dept%ROWTYPE;  -- declare record variable

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

```plsql
my_deptno := dept_rec.deptno;
```

If you declare a cursor that retrieves the last name, salary, hire date, and job title of an employee, you can use `%ROWTYPE` to declare a record that stores the same information, as follows:

```plsql
DECLARE
    CURSOR c1 IS
        SELECT ename, sal, hiredate, job FROM emp;
    emp_rec c1%ROWTYPE;  -- declare record variable that represents a row fetched from the emp table

When you execute the statement

```plsql
FETCH c1 INTO emp_rec;
```

the value in the ename column of the emp table is assigned to the ename field of emp_rec, the value in the sal column is assigned to the sal field, and so on. Figure 1–3 shows how the result might appear.

**Figure 1–3  %ROWTYPE Record**

<table>
<thead>
<tr>
<th>emp_rec.enam</th>
<th>JAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp_rec.sal</td>
<td>950.00</td>
</tr>
<tr>
<td>emp_rec.hiredate</td>
<td>03–DEC–95</td>
</tr>
<tr>
<td>emp_rec.job</td>
<td>CLERK</td>
</tr>
</tbody>
</table>
Control Structures

Control structures are the most important PL/SQL extension to SQL. Not only does PL/SQL let you manipulate Oracle data, it lets you process the data using conditional, iterative, and sequential flow-of-control statements such as IF-THEN-ELSE, CASE, FOR-LOOP, WHILE-LOOP, EXIT-WHEN, and GOTO. Collectively, these statements can handle any situation.

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; the ELSE clause defines what to do if the condition is false or null.

Consider the program below, which processes a bank transaction. Before allowing you to withdraw $500 from account 3, it makes sure the account has sufficient funds to cover the withdrawal. If the funds are available, the program debits the account. Otherwise, the program inserts a record into an audit table.

```plsql
-- available online in file 'examp2'
DECLARE
    acct_balance NUMBER(11,2);
    acct         CONSTANT NUMBER(4) := 3;
    debit_amt    CONSTANT NUMBER(5,2) := 500.00;
BEGIN
    SELECT bal INTO acct_balance FROM accounts
        WHERE account_id = acct
        FOR UPDATE OF bal;
    IF acct_balance >= debit_amt THEN
        UPDATE accounts SET bal = bal - debit_amt
            WHERE account_id = acct;
    ELSE
        INSERT INTO temp VALUES
            (acct, acct_balance, 'Insufficient funds');
            -- insert account, current balance, and message
    END IF;
    COMMIT;
END;
```

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 (which might be an entire PL/SQL block) for each case.
-- This CASE statement performs different actions based
-- on a set of conditional tests.
CASE
  WHEN shape = 'square' THEN area := side * side;
  WHEN shape = 'circle' THEN
    BEGIN
      area := pi * (radius * radius);
      DBMS_OUTPUT.PUT_LINE('Value is not exact because pi is
irrational.');
    END;
  WHEN shape = 'rectangle' THEN area := length * width;
  ELSE
    BEGIN
      DBMS_OUTPUT.PUT_LINE('No formula to calculate area of a'
shape);
      RAISE PROGRAM_ERROR;
    END;
END CASE;

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. For example, the
following loop inserts 500 numbers and their square roots into a database table:

FOR num IN 1..500 LOOP
  INSERT INTO roots VALUES (num, SQRT(num));
END LOOP;

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 the following example, you find the first employee who has a salary over $2500 and is higher in the chain of command than employee 7499:

```
-- available online in file 'examp3'
DECLARE
  salary         emp.sal%TYPE := 0;
  mgr_num        emp.mgr%TYPE;
  last_name      emp.ename%TYPE;
  starting_empno emp.empno%TYPE := 7499;
BEGIN
  SELECT mgr INTO mgr_num FROM emp
  WHERE empno = starting_empno;
  WHILE salary <= 2500 LOOP
    SELECT sal, mgr, ename INTO salary, mgr_num, last_name
    FROM emp WHERE empno = mgr_num;
  END LOOP;
  INSERT INTO temp VALUES (NULL, salary, last_name);
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    INSERT INTO temp VALUES (NULL, NULL, 'Not found');
  COMMIT;
END;
```

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

```
LOOP
...
  total := total + salary;
  EXIT WHEN total > 25000;  -- exit loop if condition is true
END LOOP;
-- control resumes here
```

**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 the following example shows:

```sql
IF rating > 90 THEN
  GOTO calc_raise;  -- branch to label
END IF;
...
<<calc_raise>>
IF job_title = 'SALESMAN' THEN  -- control resumes here
  amount := commission * 0.25;
ELSE
  amount := salary * 0.10;
END IF;
```

**Modularity**

Modularity lets you break an application down into manageable, well-defined modules. Through successive refinement, you can reduce a complex problem to a set of simple problems that have easy-to-implement solutions. PL/SQL meets this need with *program units*, which include blocks, subprograms, and packages.

**Subprograms**

PL/SQL has two types of subprograms called *procedures* and *functions*, which can take parameters and be invoked (called). As the following example shows, a subprogram is like a miniature program, beginning with a header followed by an optional declarative part, an executable part, and an optional exception-handling part:

```sql
PROCEDURE award_bonus (emp_id NUMBER) IS
    bonus        REAL;
    comm_missing EXCEPTION;
BEGIN  -- executable part starts here
    SELECT comm * 0.15 INTO bonus FROM emp WHERE empno = emp_id;
    IF bonus IS NULL THEN
        RAISE comm_missing;
    ELSE
        UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
    END IF;
EXCEPTION  -- exception-handling part starts here
    WHEN comm_missing THEN
        ...
END award_bonus;
```
When called, this procedure accepts an employee number. It uses the number to select the employee’s commission from a database table and, at the same time, compute a 15% bonus. Then, it checks the bonus amount. If the bonus is null, an exception is raised; otherwise, the employee’s payroll record is updated.

**Packages**

PL/SQL lets you bundle logically related types, variables, cursors, and subprograms into a package. Each package is easy to understand and the interfaces between packages are simple, clear, and well defined. This aids application development.

Packages usually have two parts: a specification and a body. The specification is the interface to your applications; it declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The body defines cursors and subprograms and so implements the specification.

In the following example, you package two employment procedures:

```sql
CREATE PACKAGE emp_actions AS  -- package specification
    PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
    PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS  -- package body
    PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
        BEGIN
            INSERT INTO emp VALUES (empno, ename, ...);
        END hire_employee;
    PROCEDURE fire_employee (emp_id NUMBER) IS
        BEGIN
            DELETE FROM emp WHERE empno = emp_id;
        END fire_employee;
END emp_actions;
```

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible.

Packages can be compiled and stored in an Oracle database, where their contents can be shared by many applications. When you call a packaged subprogram for the first time, the whole package is loaded into memory. So, subsequent calls to related subprograms in the package require no disk I/O. Thus, packages can enhance productivity and improve performance.
Understanding the Main Features of PL/SQL

Data Abstraction

Data abstraction lets you extract the essential properties of data while ignoring unnecessary details. Once you design a data structure, you can forget the details and focus on designing algorithms that manipulate the data structure.

Collections

The collection types TABLE and VARRAY allow you to declare index-by tables, nested tables, and variable-size arrays (varrays for short). A 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.

To reference an element, use standard subscripting syntax. For example, the following call references the fifth element in the nested table (of type Staff) returned by function new_hires:

```plsql
DECLARE
    TYPE Staff IS TABLE OF Employee;
    staffer Employee;
    FUNCTION new_hires (hiredate DATE) RETURN Staff IS
        BEGIN ... END;
BEGIN
    staffer := new_hires('10-NOV-98')(5);
    ...
END;
```

Collections work like the arrays found in most third-generation programming languages. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

Records

You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row fetched from a cursor. But, with a user-defined record, you can declare fields of your own.

Records contain uniquely named fields, which can have different datatypes. Suppose you have various data about an employee such as name, salary, and hire date. These items are dissimilar in type but logically related. A record containing a field for each item lets you treat the data as a logical unit.

Consider the following example:

```plsql
DECLARE
    ...
END;
```
TYPE TimeRec IS RECORD (hours SMALLINT, minutes SMALLINT);
TYPE MeetingTyp IS RECORD (date_held DATE, duration TimeRec, location VARCHAR2(20), purpose VARCHAR2(50));

Notice that you can nest records. That is, a record can be a component of another record.

**Object Types**

In PL/SQL, object-oriented programming is based on object types. An object type encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called attributes. The functions and procedures that characterize the behavior of the object type are called 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.

When you define an object type using the `CREATE TYPE` statement (in SQL*Plus for example), you create an abstract template for some real-world object. As the following example of a bank account shows, the template specifies only those attributes and behaviors the object will need in the application environment:

```sql
CREATE TYPE Bank_Account AS OBJECT (acct_number INTEGER(5), balance REAL, status VARCHAR2(10),
  MEMBER PROCEDURE open (amount IN REAL),
  MEMBER PROCEDURE verify_acct (num IN INTEGER),
  MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL),
  MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL),
  MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL),
  MEMBER FUNCTION curr_bal (num IN INTEGER) RETURN REAL);
```

At run time, when the data structure is filled with values, you have created an instance of an abstract bank account. You can create as many instances (called objects) as you need. Each object has the number, balance, and status of an actual bank account.
Information Hiding

With information hiding, you see only the details that are relevant at a given level of algorithm and data structure design. Information hiding keeps high-level design decisions separate from low-level design details, which are more likely to change.

Algorithms

You implement information hiding for algorithms through top-down design. Once you define the purpose and interface specifications of a low-level procedure, you can ignore the implementation details. They are hidden at higher levels. For example, the implementation of a procedure named `raise_salary` is hidden. All you need to know is that the procedure will increase a specific employee’s salary by a given amount. Any changes to the definition of `raise_salary` are transparent to calling applications.

Data Structures

You implement information hiding for data structures through data encapsulation. By developing a set of utility subprograms for a data structure, you insulate it from users and other developers. That way, other developers know how to use the subprograms that operate on the data structure but not how the structure is represented.

With PL/SQL packages, you can specify whether subprograms are public or private. Thus, packages enforce data encapsulation by letting you put subprogram definitions in a black box. A private definition is hidden and inaccessible. Only the package, not your application, is affected if the definition changes. This simplifies maintenance and enhancement.

Error Handling

PL/SQL makes it easy to detect and process predefined and user-defined error conditions called exceptions. 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. To handle raised exceptions, you write separate routines called exception handlers.

Predefined exceptions are raised implicitly by the runtime system. For example, if you try to divide a number by zero, PL/SQL raises the predefined exception `ZERO_DIVIDE` automatically. You must raise user-defined exceptions explicitly with the `RAISE` statement.
You can define exceptions of your own in the declarative part of any PL/SQL block or subprogram. In the executable part, you check for the condition that needs special attention. If you find that the condition exists, you execute a `RAISE` statement. In the example below, you compute the bonus earned by a salesperson. The bonus is based on salary and commission. So, if the commission is null, you raise the exception `comm_missing`.

```plsql
DECLARE
  comm_missing EXCEPTION;  -- declare exception
BEGIN
  ...
  IF commission IS NULL THEN
    RAISE comm_missing;  -- raise exception
  END IF;
  bonus := (salary * 0.10) + (commission * 0.15);
EXCEPTION
  WHEN comm_missing THEN ... -- process the exception
```

**PL/SQL Architecture**

The PL/SQL compilation and run-time system is a technology, not an independent product. Think of this technology as an engine that compiles and executes PL/SQL blocks and subprograms. The engine can be installed in an Oracle server or in an application development tool such as Oracle Forms or Oracle Reports. So, PL/SQL can reside in two environments:

- The Oracle database server
- Oracle tools

These two environments are independent. PL/SQL is bundled with the Oracle server but might be unavailable in some tools. In either environment, the PL/SQL engine accepts as input any valid PL/SQL block or subprogram. Figure 1–4 shows the PL/SQL engine processing an anonymous block. The engine executes procedural statements but sends SQL statements to the SQL Statement Executor in the Oracle server.
In the Oracle Database Server

Application development tools that lack a local PL/SQL engine must rely on Oracle to process PL/SQL blocks and subprograms. When it contains the PL/SQL engine, an Oracle server can process PL/SQL blocks and subprograms as well as single SQL statements. The Oracle server passes the blocks and subprograms to its local PL/SQL engine.

Anonymous Blocks

Anonymous PL/SQL blocks can be embedded in an Oracle Precompiler or OCI program. At run time, the program, lacking a local PL/SQL engine, sends these blocks to the Oracle server, where they are compiled and executed. Likewise, interactive tools such as SQL*Plus and Enterprise Manager, lacking a local PL/SQL engine, must send anonymous blocks to Oracle.

Stored Subprograms

Subprograms can be compiled separately and stored permanently in an Oracle database, ready to be executed. A subprogram explicitly \texttt{CREATE}d using an Oracle tool is called a \textit{stored} subprogram. Once compiled and stored in the data dictionary, it is a schema object, which can be referenced by any number of applications connected to that database.
Stored subprograms defined within a package are called *packaged* subprograms. Those defined independently are called *standalone* subprograms. Those defined within another subprogram or within a PL/SQL block are called *local* subprograms, which cannot be referenced by other applications and exist only for the convenience of the enclosing block.

Stored subprograms offer higher productivity, better performance, memory savings, application integrity, and tighter security. For example, by designing applications around a library of stored procedures and functions, you can avoid redundant coding and increase your productivity.

You can call stored subprograms from a database trigger, another stored subprogram, an Oracle Precompiler application, an OCI application, or interactively from SQL*Plus or Enterprise Manager. For example, you might call the standalone procedure `create_dept` from SQL*Plus as follows:

```sql
SQL> CALL create_dept('FINANCE', 'NEW YORK');
```

Subprograms are stored in parsed, compiled form. So, when called, they are loaded and passed to the PL/SQL engine immediately. Also, they take advantage of shared memory. So, only one copy of a subprogram need be loaded into memory for execution by multiple users.

**Database Triggers**

A database trigger is a stored subprogram associated with a database table, view, or event. For instance, you can have Oracle fire a trigger automatically before or after an INSERT, UPDATE, or DELETE statement affects a table. One of the many uses for database triggers is to audit data modifications. For example, the following table-level trigger fires whenever salaries in the `emp` table are updated:

```sql
CREATE TRIGGER audit_sal
  AFTER UPDATE OF sal ON emp
  FOR EACH ROW
BEGIN
  INSERT INTO emp_audit VALUES ...;
END;
```

The executable part of a trigger can contain procedural statements as well as SQL data manipulation statements. Besides table-level triggers, there are *instead-of* triggers for views and *system-event* triggers for schemas. For more information, see *Oracle9i Application Developer’s Guide - Fundamentals*.
Advantages of PL/SQL

In Oracle Tools

When it contains the PL/SQL engine, an application development tool can process PL/SQL blocks and subprograms. The tool passes the blocks to its local PL/SQL engine. The engine executes all procedural statements at the application site and sends only SQL statements to Oracle. Thus, most of the work is done at the application site, not at the server site.

Furthermore, if the block contains no SQL statements, the engine executes the entire block at the application site. This is useful if your application can benefit from conditional and iterative control.

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

Advantages of PL/SQL

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

- Support for SQL
- Support for object-oriented programming
- Better performance
- Higher productivity
- Full portability
- Tight integration with Oracle
- Tight security

Support for SQL

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

SQL is non-procedural, meaning that you can state what you want done without stating how to do it. Oracle determines the best way to carry out your request. There is no necessary connection between consecutive statements because Oracle executes SQL statements one at a time.
PL/SQL lets you use all the SQL data manipulation, cursor control, and transaction control commands, as well as all the SQL functions, operators, and pseudocolumns. So, you can manipulate Oracle data flexibly and safely. Also, PL/SQL fully supports SQL datatypes. That reduces the need to convert data passed between your applications and the database.

PL/SQL also supports dynamic SQL, an advanced programming technique that makes your applications more flexible and versatile. Your programs can build and process SQL data definition, data control, and session control statements “on the fly” at run time.

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 allowing 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. That helps your programs better reflect the world they are trying to simulate.

Better Performance

Without PL/SQL, Oracle must process SQL statements one at a time. Each SQL statement results in another call to Oracle and higher performance overhead. In a networked environment, the overhead can become significant. Every time a SQL statement is issued, it must be sent over the network, creating more traffic.

However, with PL/SQL, an entire block of statements can be sent to Oracle at one time. This can drastically reduce communication between your application and Oracle. As Figure 1–5 shows, if your application is database intensive, you can use PL/SQL blocks and subprograms to group SQL statements before sending them to Oracle for execution.
PL/SQL stored procedures are compiled once and stored in executable form, so procedure calls are quick and efficient. Also, stored procedures, which execute in the server, can be invoked over slow network connections with a single call. That reduces network traffic and improves round-trip response times. Executable code is automatically cached and shared among users. That lowers memory requirements and invocation overhead.

**Figure 1–5  PL/SQL Boosts Performance**

PL/SQL also improves performance by adding procedural processing power to Oracle tools. Using PL/SQL, a tool can do any computation quickly and efficiently without calling on the Oracle server. This saves time and reduces network traffic.

**Higher Productivity**

PL/SQL adds functionality to non-procedural tools such as Oracle Forms and Oracle Reports. With PL/SQL in these tools, you can use familiar procedural constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger. You need not use multiple trigger steps, macros, or user exits. Thus, PL/SQL increases productivity by putting better tools in your hands.
Also, PL/SQL is the same in all environments. As soon as you master PL/SQL with one Oracle tool, you can transfer your knowledge to other tools, and so multiply the productivity gains. For example, scripts written with one tool can be used by other tools.

**Full Portability**

Applications written in PL/SQL are portable to any operating system and platform on which Oracle runs. In other words, PL/SQL programs can run anywhere Oracle can run; you need not tailor them to each new environment. That means you can write portable program libraries, which can be reused in different environments.

**Tight Integration with SQL**

The PL/SQL and SQL languages are tightly integrated. PL/SQL supports all the SQL datatypes and the non-value **null**. That allows you manipulate Oracle data easily and efficiently. It also helps you to write high-performance code.

The **%type** and **%rowtype** attributes further integrate PL/SQL with SQL. For example, you can use the **%type** attribute to declare variables, basing the declarations on the definitions of database columns. If a definition changes, the variable declaration changes accordingly the next time you compile or run your program. The new definition takes effect without any effort on your part. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.

**Tight Security**

PL/SQL stored procedures enable you to partition application logic between the client and server. That way, you can prevent client applications from manipulating sensitive Oracle data. Database triggers written in PL/SQL can disable application updates selectively and do content-based auditing of user inserts.

Furthermore, you can restrict access to Oracle data by allowing users to manipulate it only through stored procedures that execute with their definer’s privileges. For example, you can grant users access to a procedure that updates a table, but not grant them access to the table itself.
Advantages of PL/SQL
There are six essentials in painting. The first is called spirit; the second, rhythm; the third, thought; the fourth, scenery; the fifth, the brush; and the last is the ink. —Ching Hao

The previous chapter provided an overview of PL/SQL. This chapter focuses on the small-scale aspects of the language. Like every other programming language, PL/SQL has a character set, reserved words, punctuation, datatypes, rigid syntax, and fixed rules of usage and statement formation. You use these basic elements of PL/SQL to represent real-world objects and operations.

This chapter discusses the following topics:
- Character Set
- Lexical Units
- Declarations
- Naming Conventions
- Scope and Visibility
- Assignments
- Expressions and Comparisons
- Built-In Functions
Character Set

You write a PL/SQL program as lines of text using a specific set of characters. The PL/SQL character set includes

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

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

Lexical Units

A line of PL/SQL text contains groups of characters known as *lexical units*, which can be classified as follows:

- delimiters (simple and compound symbols)
- identifiers, which include reserved words
- literals
- comments

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

```
IF x > y THEN high := x; ENDIF;  -- not allowed
```

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

```
count := count + 1;  -- not allowed
```

To show structure, you can divide lines using carriage returns and indent lines using spaces or tabs. Compare these IF statements for readability:

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

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

A *delimiter* is a simple or compound symbol that has a special meaning to PL/SQL. For example, you use delimiters to represent arithmetic operations such as addition and subtraction. Simple symbols consist of one character. A list follows:

<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>
</tbody>
</table>
Compound symbols consist of two characters. A list follows:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>&lt;=</td>
<td>relational operator</td>
</tr>
<tr>
<td>=&gt;</td>
<td>relational operator</td>
</tr>
<tr>
<td>--</td>
<td>single-line comment indicator</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. Some examples of identifiers follow:

X
t2
phone#
credit_limit
LastName
oracle$number
An identifier consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs. Other characters such as hyphens, slashes, and spaces are not allowed, as the following examples show:

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

The next examples show that adjoining and trailing dollar signs, underscores, and number signs are allowed:

- money$$tree
- SN##
- try_again_

You can use upper, lower, or mixed case to write identifiers. PL/SQL is not case sensitive except within string and character literals. So, if the only difference between identifiers is the case of corresponding letters, PL/SQL considers the identifiers to be the same, as the following example shows:

- lastname
- LastName
- LASTNAME

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

- lastname
- last_name

Identifiers should be descriptive. So, avoid obscure names such as cpm. Instead, use meaningful names such as cost_per_thousand.
Reserved Words

Some identifiers, called reserved words, have a special syntactic meaning to PL/SQL and so should not be redefined. For example, the words BEGIN and END, which bracket the executable part of a block or subprogram, are reserved. As the next example shows, if you try to redefine a reserved word, you get a compilation error:

```
DECLARE
  end BOOLEAN;  -- not allowed; causes compilation error
```

However, you can embed reserved words in an identifier, as the following example shows:

```
DECLARE
  end_of_game BOOLEAN;  -- allowed
```

Often, reserved words are written in upper case to promote readability. However, like other PL/SQL identifiers, reserved words can be written in lower or mixed case. For a list of reserved words, see Appendix F.

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.
Some PL/SQL reserved words are not reserved by SQL. For example, you can use the PL/SQL reserved word `TYPE` in a `CREATE TABLE` statement to name a database column. But, if a SQL statement in your program refers to that column, you get a compilation error, as the following example shows:

```sql
SELECT acct, type, bal INTO ... -- causes compilation error
```

To prevent the error, enclose the uppercase column name in double quotes, as follows:

```sql
SELECT acct, "TYPE", bal INTO ...
```

The column name cannot appear in lower or mixed case (unless it was defined that way in the `CREATE TABLE` statement). For example, the following statement is invalid:

```sql
SELECT acct, "type", bal INTO ... -- causes compilation error
```

Alternatively, you can create a view that renames the troublesome column, then use the view instead of the base table in SQL statements.

---

**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.

**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. Some examples follow:

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

A real literal is an optionally signed whole or fractional number with a decimal point. Several examples follow:

```plaintext
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.
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. A few examples follow:

\[
2E5 \quad 1.0E-7 \quad 3.14159e0 \quad -1E38 \quad -9.5e-3
\]

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

\[
5E3 = 5 \times 10**3 = 5 \times 1000 = 5000
\]

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

\[
5E-3 = 5 \times 10**-3 = 5 \times 0.001 = 0.005
\]

As the following example shows, if the value of a numeric literal falls outside the
range 1E-130 .. 10E125, you get a compilation error:

```
DECLARE
    n NUMBER;
BEGIN
    n := 10E127;  -- causes a 'numeric overflow or underflow' error
```

**Character Literals**

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

```
'Z'  'S'  '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. Several examples follow:

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

All string literals except the null string ("") have datatype CHAR.

Given that apostrophes (single quotes) delimit string literals, how do you represent an apostrophe within a string? As the next example shows, you write two single quotes, which is not the same as writing a double quote:

'Don’t leave without saving your work.’

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

'baker'
'Baker'

Boolean Literals

Boolean literals are the predefined values TRUE, FALSE, and NULL (which 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.

Datetime Literals

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

```
DECLARE
  d1 DATE := DATE '1998-12-25';
  t1 TIMESTAMP := TIMESTAMP '1997-10-22 13:01:01';
  t2 TIMESTAMP WITH TIME ZONE := TIMESTAMP '1997-01-31 09:26:56.66
  +02:00';
  -- Three years and two months
  -- (For greater precision, we would use the day-to-second interval)
  i1 INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO MONTH;
  -- Five days, four hours, three minutes, two and 1/100 seconds
  i2 INTERVAL DAY TO SECOND := INTERVAL '5 04:03:02.01' DAY TO SECOND;
...
You can also specify whether a given interval value is \texttt{YEAR TO MONTH} or \texttt{DAY TO SECOND}. For example, \texttt{current_timestamp} \texttt{- current_timestamp} produces a value of type \texttt{INTERVAL DAY TO SECOND} by default. You can specify the type of the interval using the formats:

- \texttt{(interval\_expression) DAY TO SECOND}
- \texttt{(interval\_expression) YEAR TO MONTH}

For details on the syntax for the date and time types, see the \textit{Oracle9i SQL Reference}. For examples of performing date/time arithmetic, see \textit{Oracle9i Application Developer’s Guide - Fundamentals}.

### Comments

The PL/SQL compiler ignores comments, but you should not. Adding comments to your program promotes readability and aids understanding. Generally, you use comments to describe the purpose and use of each code segment. PL/SQL supports two comment styles: single-line and multi-line.

#### Single-Line Comments

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

```plsql
-- begin processing
SELECT sal INTO salary FROM emp -- get current salary
    WHERE empno = emp_id;
bonus := salary * 0.15; -- compute bonus amount
```

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

While testing or debugging a program, you might want to disable a line of code. The following example shows how you can "comment-out" the line:

```plsql
-- DELETE FROM emp WHERE comm IS NULL;
```

#### Multi-line Comments

Multi-line comments begin with a slash-asterisk (\texttt{/*}), end with an asterisk-slash (\texttt{*/}), and can span multiple lines. Some examples follow:

```plsql
BEGIN
    ...
    /* Compute a 15\% bonus for top-rated employees. */
    IF rating > 90 THEN
```
bonus := salary * 0.15 /* bonus is based on salary */
ELSE
  bonus := 0;
END IF;
...
/* The following line computes the area of a
circle using pi, which is the ratio between
the circumference and diameter. */
area := pi * radius**2;
END;

You can use multi-line comment delimiters to comment-out whole sections of code, as the following example shows:

/*
 LOOP
  FETCH c1 INTO emp_rec;
  EXIT WHEN c1%NOTFOUND;
  ...
 END LOOP;
 */

Restrictions on Comments
You cannot nest comments. Also, you cannot use single-line comments in a PL/SQL block that will be processed dynamically by an Oracle Precompiler program because end-of-line characters are ignored. As a result, single-line comments extend to the end of the block, not just to the end of a line. So, use multi-line comments instead.

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

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

A couple of examples follow:
birthday  DATE;
emp_count SMALLINT := 0;
The first declaration names a variable of type `DATE`. The second declaration names a variable of type `SMALLINT` and uses the assignment operator to assign an initial value of zero to the variable.

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

```sql
pi     REAL := 3.14159;
radius REAL := 1;
area   REAL := pi * radius**2;
```

By default, variables are initialized to `NULL`. So, these declarations are equivalent:

```sql
birthday DATE;
birthday DATE := NULL;
```

In the declaration of a constant, the keyword `CONSTANT` must precede the type specifier, as the following example shows:

```sql
credit_limit CONSTANT REAL := 5000.00;
```

This declaration names a constant of type `REAL` and assigns an initial (also final) value of 5000 to the constant. A constant must be initialized in its declaration. Otherwise, you get a compilation error when the declaration is elaborated. (The processing of a declaration by the PL/SQL compiler is called *elaboration*.)

### Using DEFAULT

You can use the keyword `DEFAULT` instead of the assignment operator to initialize variables. For example, the declaration

```sql
blood_type CHAR := 'O';
```

can be rewritten as follows:

```sql
blood_type CHAR DEFAULT 'O';
```

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. A couple of examples follow:

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

You can also use `DEFAULT` to initialize subprogram parameters, cursor parameters, and fields in a user-defined record.
Using NOT NULL

Besides assigning an initial value, declarations can impose the NOT NULL constraint, as the following example shows:

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

You cannot assign nulls to a variable defined as NOT NULL. If you try, PL/SQL raises the predefined exception VALUE_ERROR. The NOT NULL constraint must be followed by an initialization clause. For example, the following declaration is not allowed:

```sql
acct_id INTEGER(5) NOT NULL;  -- not allowed; not initialized
```

PL/SQL provide subtypes NATURALN and POSITIVEN that are predefined as NOT NULL. For instance, the following declarations are equivalent:

```sql
emp_count NATURAL NOT NULL := 0;
emp_count NATURALN := 0;
```

In NATURALN and POSITIVEN declarations, the type specifier must be followed by an initialization clause. Otherwise, you get a compilation error. For example, the following declaration is not allowed:

```sql
line_items POSITIVEN;  -- not allowed; not initialized
```

Using %TYPE

The %TYPE attribute provides the datatype of a variable or database column. In the following example, %TYPE provides the datatype of a variable:

```sql
credit REAL(7,2);
debit  credit%TYPE;
```

Variables declared using %TYPE are treated like those declared using a datatype specifier. For example, given the previous declarations, PL/SQL treats debit like a REAL(7,2) variable. The next example shows that a %TYPE declaration can include an initialization clause:

```sql
balance         NUMBER(7,2);
minimum_balance balance%TYPE := 10.00;
```
The `%TYPE` attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference an owner, table, and column, as in:

```
my_dname scott.dept.dname%TYPE;
```

Using `%TYPE` to declare `my_dname` has two advantages. First, you need not know the exact datatype of `dname`. Second, if the database definition of `dname` changes, the datatype of `my_dname` changes accordingly at run time.

However, `%TYPE` variables do not inherit the `NOT NULL` column constraint. In the next example, even though the database column `empno` is defined as `NOT NULL`, you can assign a null to the variable `my_empno`:

```
DECLARE
  my_empno emp.empno%TYPE;
...
BEGIN
  my_empno := NULL;  -- this works
```

**Using `%ROWTYPE`**

The `%ROWTYPE` attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table or fetched from a cursor or strongly typed cursor variable. In the example below, you declare two records. The first record stores a row selected from the `emp` table. The second record stores a row fetched from cursor `c1`.

```
DECLARE
  emp_rec emp%ROWTYPE;
  CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
  dept_rec c1%ROWTYPE;

Columns in a row and corresponding fields in a record have the same names and datatypes. However, fields in a `%ROWTYPE` record do not inherit the `NOT NULL` column constraint.

In the following example, you select column values into record `emp_rec`:

```
BEGIN
  SELECT * INTO emp_rec FROM emp WHERE ...
```
The column values returned by the SELECT statement are stored in fields. To reference a field, you use dot notation. For example, you might reference the deptno field as follows:

```sql
IF emp_rec.deptno = 20 THEN ...
```

Also, you can assign the value of an expression to a specific field, as the following examples show:

```sql
emp_rec.ename := ’JOHNSON’;
emp_rec.sal := emp_rec.sal * 1.15;
```

In the final example, you use %ROWTYPE to define a packaged cursor:

```sql
CREATE PACKAGE emp_actions AS
  CURSOR c1 RETURN emp%ROWTYPE;  -- declare cursor specification
  ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
  CURSOR c1 RETURN emp%ROWTYPE IS  -- define cursor body
    SELECT * FROM emp WHERE sal > 3000;
  ...
END emp_actions;
```

**Aggregate Assignment**

A %ROWTYPE declaration cannot include an initialization clause. However, there are two ways to assign values to all fields in a record at once. First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor. For example, the following assignment is allowed:

```sql
DECLARE
  dept_rec1 dept%ROWTYPE;
  dept_rec2 dept%ROWTYPE;
  CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
  dept_rec3 c1%ROWTYPE;
BEGIN
  ...
  dept_rec1 := dept_rec2;
END;
```

However, because dept_rec2 is based on a table and dept_rec3 is based on a cursor, the following assignment is not allowed:

```sql
depth_rec2 := dept_rec3;  -- not allowed
```
Second, you can assign a list of column values to a record by using the SELECT or FETCH statement, as the example below shows. The column names must appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement.

```plsql
DECLARE
  dept_rec dept%ROWTYPE;
  ...
BEGIN
  SELECT * INTO dept_rec FROM dept WHERE deptno = 30;
  ...
END;
```

However, you cannot assign a list of column values to a record by using an assignment statement. So, the following syntax is not allowed:

```plsql
record_name := (value1, value2, value3, ...);  -- not allowed
```

### Using Aliases

Select-list items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases. In the following example, you use an alias called wages:

```plsql
-- available online in file 'examp4'
DECLARE
  CURSOR my_cursor IS
    SELECT sal + NVL(comm, 0) wages, ename FROM emp;
  my_rec  my_cursor%ROWTYPE;
BEGIN
  OPEN my_cursor;
  LOOP
    FETCH my_cursor INTO my_rec;
    EXIT WHEN my_cursor%NOTFOUND;
    IF my_rec.wages > 2000 THEN
      INSERT INTO temp VALUES (NULL, my_rec.wages, my_rec.ename);
    END IF;
  END LOOP;
  CLOSE my_cursor;
END;
```
Restrictions on Declarations

PL/SQL does not allow forward references. You must declare a variable or constant before referencing it in other statements, including other declarative statements. For example, the following declaration of maxi is not allowed:

```sql
maxi INTEGER := 2 * mini; -- not allowed
mini INTEGER := 15;
```

However, PL/SQL does allow the forward declaration of subprograms. For more information, see "Declaring PL/SQL Subprograms" on page 8-10.

Some languages allow you to declare a list of variables that have the same datatype. PL/SQL does not allow this. For example, the following declaration is not allowed:

```sql
i, j, k SMALLINT; -- not allowed
```

You must declare each variable separately:

```sql
i SMALLINT;
j SMALLINT;
k SMALLINT;
```

Naming Conventions

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

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

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

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

Scoping

Within the same scope, all declared identifiers must be unique. So, even if their datatypes differ, variables and parameters cannot share the same name. In the following example, the second declaration is not allowed:

```plsql
DECLARE
    valid_id BOOLEAN;
    valid_id VARCHAR2(5);  -- not allowed duplicate identifier
```

For the scoping rules that apply to identifiers, see "Scope and Visibility" on page 2-19.

Case Sensitivity

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

```plsql
DECLARE
    zip_code INTEGER;
    Zip_Code INTEGER;  -- same as zip_code
```

Name Resolution

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables and formal parameters. For example, the following DELETE statement removes all employees from the emp table, not just 'KING', because Oracle assumes that both enames in the WHERE clause refer to the database column:

```plsql
DECLARE
    ename VARCHAR2(10) := 'KING';
BEGIN
    DELETE FROM emp WHERE ename = ename;
    ...
```
In such cases, to avoid ambiguity, prefix the names of local variables and formal parameters with my_, as follows:

```
DECLARE
    my_ename VARCHAR2(10);
```

Or, use a block label to qualify references, as in

```
<<main>>
DECLARE
    ename VARCHAR2(10) := 'KING';
BEGIN
    DELETE FROM emp WHERE ename = main.ename;
    ...
```

The next example shows that you can use a subprogram name to qualify references to local variables and formal parameters:

```
FUNCTION bonus (deptno IN NUMBER, ...) RETURN REAL IS
    job CHAR(10);
BEGIN
    SELECT ... WHERE deptno = bonus.deptno AND job = bonus.job;
    ...
```

For a full discussion of name resolution, see Appendix D.

Scope and Visibility

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

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

Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks. The two items represented by the identifier are distinct, and any change in one does not affect the other. However, a block cannot reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.
The example below illustrates the scope rules. Notice that the identifiers declared in one sub-block cannot be referenced in the other sub-block. That is because a block cannot reference identifiers declared in other blocks nested at the same level.

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

Recall that global identifiers can be redeclared in a sub-block, in which case the local declaration prevails and the sub-block cannot reference the global identifier unless you use a qualified name. The qualifier can be the label of an enclosing block, as the following example shows:
<<outer>>
DECLARE
    birthdate DATE;
BEGIN
    DECLARE
        birthdate DATE;
    BEGIN
        ...
        IF birthdate = outer.birthdate THEN ...
    END;
    ...
END;

As the next example shows, the qualifier can also be the name of an enclosing subprogram:
PROCEDURE check_credit (...) IS
    rating NUMBER;
FUNCTION valid (...) RETURN BOOLEAN IS
    rating NUMBER;
BEGIN
    ...
    IF check_credit.rating < 3 THEN ...
END;
BEGIN
    ...
END;

However, within the same scope, a label and a subprogram cannot have the same name.
Assignments

Variables and constants are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL. So, unless you expressly initialize a variable, its value is undefined, as the following example shows:

```sql
DECLARE
    count INTEGER;
...
BEGIN
    count := count + 1;  -- assigns a null to count
```

The expression on the right of the assignment operator yields NULL because `count` is null. To avoid unexpected results, never reference a variable before you assign it a value.

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

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

The expression following the assignment operator can be arbitrarily complex, but it must yield a datatype that is the same as or convertible to the datatype of the variable.

Boolean Values

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. For example, given the declaration

```sql
DECLARE
    done BOOLEAN;
```

the following statements are allowed:

```sql
BEGIN
    done := FALSE;
    WHILE NOT done LOOP
        ...
    END LOOP;
```

When applied to an expression, the relational operators return a Boolean value. So, the following assignment is allowed:

```sql
done := (count > 500);
```
Database Values

You can use the `SELECT` statement to have Oracle assign values to a variable. For each item in the select list, there must be a corresponding, type-compatible variable in the `INTO` list. An example follows:

```plsql
DECLARE
    emp_id   emp.empno%TYPE;
    emp_name emp.ename%TYPE;
    wages    NUMBER(7,2);
BEGIN
    -- assign a value to emp_id here
    SELECT ename, sal + comm
        INTO emp_name, wages
        FROM emp
        WHERE empno = emp_id;
    ...
END;
```

However, you cannot select column values into a Boolean variable.

Expressions and Comparisons

Expressions are constructed using operands and operators. An operand is a variable, constant, literal, or function call that contributes a value to an expression. An example of a simple arithmetic expression follows:

```
-X / 2 + 3
```

Unary operators such as the negation operator (`-`) operate on one operand; binary operators such as the division operator (`/`) operate on two operands. PL/SQL has no ternary operators.

The simplest expressions consist of a single variable, which yields a value directly. PL/SQL evaluates (finds the current value of) an expression by combining the values of the operands in ways specified by the operators. This always yields a single value and datatype. PL/SQL determines the datatype by examining the expression and the context in which it appears.
Operator Precedence

The operations within an expression are done in a particular order depending on their precedence (priority). Table 2–1 shows the default order of operations from first to last (top to bottom).

Table 2–1 Order of Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>+, −</td>
<td>identity, negation</td>
</tr>
<tr>
<td>*, /</td>
<td>multiplication, division</td>
</tr>
<tr>
<td>+, −,</td>
<td></td>
</tr>
<tr>
<td>=, &lt;&gt;, &lt;=, &gt;=, !=, ~=, ^=</td>
<td>comparison, IS NULL, LIKE, BETWEEN, IN</td>
</tr>
<tr>
<td>NOT</td>
<td>logical negation</td>
</tr>
<tr>
<td>AND</td>
<td>conjunction</td>
</tr>
<tr>
<td>OR</td>
<td>inclusion</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
5 + 12 / 4 \\
12 / 4 + 5
\end{align*}
\]

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

\[
(8 + 6) / 2
\]

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

\[
100 + (20 / 5 + (7 - 3))
\]

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

\[
(salary * 0.05) + (commission * 0.25)
\]
Logical Operators

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

Table 2–2 Logic Truth Table

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x AND y</th>
<th>x OR y</th>
<th>NOT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
<td>TRUE</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>FALSE</td>
<td>FALSE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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

NOT NULL returns NULL because nulls are indeterminate. It follows that if you apply the NOT operator to a null, the result is also indeterminate. Be careful. Nulls can cause unexpected results; see "Handling Null Values in Comparisons and Conditional Statements" on page 2-33.

Order of Evaluation

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

NOT (valid AND done) | NOT valid AND done

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

(NOT valid) AND done
In the following example, notice that when valid has the value FALSE, the whole expression yields FALSE regardless of the value of done:

valid AND done

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

valid OR done

**Short-Circuit Evaluation** When evaluating a logical expression, PL/SQL uses short-circuit evaluation. That is, PL/SQL stops evaluating the expression as soon as the result can be determined. This lets you write expressions that might otherwise cause an error. Consider the following OR expression:

```plsql
DECLARE
  on_hand  INTEGER;
  on_order INTEGER;
BEGIN
  ...
  IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
    ...
  END IF;
END;
```

When the value of on_hand is zero, the left operand yields TRUE, so PL/SQL need not evaluate the right operand. If PL/SQL were to evaluate both operands before applying the OR operator, the right operand would cause a division by zero error. In any case, it is a poor programming practice to rely on short-circuit evaluation.

**Comparison Operators**

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

```plsql
IF quantity_on_hand > 0 THEN
  UPDATE inventory SET quantity = quantity - 1
  WHERE part_number = item_number;
ELSE
  ...
END IF;
```
Relational Operators
The relational operators allow you to compare arbitrarily complex expressions. The following list gives the meaning of each operator:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;, !=, ~=, ^=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

IS NULL Operator
The IS NULL operator returns the Boolean value TRUE if its operand is null or FALSE if it is not null. Comparisons involving nulls always yield NULL. So, test for nullity (the state of being null), as follows:

IF variable IS NULL THEN ... 

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

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

ename LIKE 'J%SON'

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

45 BETWEEN 38 AND 44
Expressions and Comparisons

**IN Operator**

The **IN** operator tests set membership. It means "equal to any member of." The set can contain nulls, but they are ignored. For example, the following statement does not delete rows in which the `ename` column is null:

```
DELETE FROM emp WHERE ename IN (NULL, 'KING', 'FORD');
```

Furthermore, expressions of the form

```
value NOT IN set
```

yield FALSE if the set contains a null. For example, instead of deleting rows in which the `ename` column is not null and not 'KING', the following statement deletes no rows:

```
DELETE FROM emp WHERE ename NOT IN (NULL, 'KING');
```

**Concatenation Operator**

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

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

returns the following value:

```
'suitcase'
```

If both operands have datatype CHAR, the concatenation operator returns a CHAR value. If either operand is a CLOB value, the operator returns a temporary CLOB. Otherwise, it returns a VARCHAR2 value.

**Boolean Expressions**

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

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

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

the following expression is true:

```plsql
number1 > number2
```

Boolean Character Expressions
You can compare character values for equality or inequality. By default, comparisons are based on the binary values of each byte in the string.

For example, given the assignments

```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 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.

There are semantic differences between the CHAR and VARCHAR2 base types that come into play when you compare character values. For more information, see **Appendix B**.

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 "Character Types" on page 3-5.
Boolean Date Expressions

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

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

the following expression is true:

date1 > date2

Guidelines for PL/SQL Boolean Expressions

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

```
count := 1;
IF count = 1.0 THEN
  ...
END IF;
```

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

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

The debugged version follows:

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

■ A Boolean variable is itself either true or false. So, comparisons with the
Boolean values TRUE and FALSE are redundant. For example, assuming the
variable done is of type BOOLEAN, the WHILE statement

```
WHILE NOT (done = TRUE) LOOP
  ...
END LOOP;
```

can be simplified as follows:

```
WHILE NOT done LOOP
  ...
END LOOP;
```
Expressions and Comparisons

- Using CLOB values with comparison operators, or functions such as LIKE and BETWEEN, can result in creation of temporary LOBs. You might need to make sure your temporary tablespace is large enough to handle these temporary LOBs. For details, see the "Modeling and Design" chapter in Oracle9i Application Developer's Guide - Large Objects (LOBs).

CASE Expressions

A CASE expression selects a result and returns it. To select the result, the CASE expression uses a selector, an expression whose value is used to select one of several alternatives. A CASE expression has the following form:

```
CASE selector
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
```

The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. The first WHEN clause that matches the value of the selector determines the result value, and subsequent WHEN clauses are not evaluated. An example follows:

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

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

An alternative to the CASE expression is the CASE statement, where each WHEN clause can be an entire PL/SQL block. For details, see “CASE Statement” on page 4-5.

**Searched CASE Expression**

PL/SQL also provides a searched CASE expression, which has the form:

```
CASE
  WHEN search_condition1 THEN result1
  WHEN search_condition2 THEN result2
  ...
  WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

A searched CASE expression has no selector. Each WHEN clause contains a search condition that yields a Boolean value, which lets you test different variables or multiple conditions in a single WHEN clause. An example follows:

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

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

Handling Null Values in Comparisons and Conditional Statements

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

- Comparisons involving nulls always yield NULL
- Applying the logical operator NOT to a null yields NULL
- In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed
- If the expression in a simple CASE statement or CASE expression yields NULL, it cannot be matched by using WHEN NULL. In this case, you would need to use the searched case syntax and test WHEN expression IS NULL.

In the example below, you might expect the sequence of statements to execute because \( x \) and \( y \) seem unequal. But, nulls are indeterminate. Whether or not \( x \) is equal to \( y \) is unknown. Therefore, the IF condition yields NULL and the sequence of statements is bypassed.

```plsql
x := 5;
y := NULL;
...
IF x != y THEN -- yields NULL, not TRUE
  sequence_of_statements; -- not executed
END IF;
```

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

```plsql
a := NULL;
b := NULL;
...
IF a = b THEN -- yields NULL, not TRUE
  sequence_of_statements; -- not executed
END IF;
```

NOT Operator

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

```plsql
IF (aluminum > 3) OR (aluminum < 3) THEN
  sequence_of_statements; -- not executed
END IF;
```
IF x > y THEN    |    IF NOT x > y THEN
  high := x;     |    high := y;
ELSE              | ELSE
  high := y;     |    high := x;
END IF;           | END IF;

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

Zero-Length Strings

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

null_string := TO_CHAR('');
zip_code := SUBSTR(address, 25, 0);
valid := (name != '');

So, use the IS NULL operator to test for null strings, as follows:

IF my_string IS NULL THEN ...

Concatenation Operator

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

‘apple’ || NULL || NULL || ‘sauce’

returns the following value:

‘applesauce’

Functions

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

The function DECODE compares its first argument to one or more search expressions, which are paired with result expressions. Any search or result expression can be null. If a search is successful, the corresponding result is returned. In the following example, if the column rating is null, DECODE returns the value 1000:

SELECT DECODE(rating, NULL, 1000, 'C', 2000, 'B', 4000, 'A', 5000)
INTO credit_limit FROM accts WHERE acctno = my_acctno;

The function NVL returns the value of its second argument if its first argument is null. In the example below, if hire_date is null, NVL returns the value of SYSDATE. Otherwise, NVL returns the value of hire_date:

start_date := NVL(hire_date, SYSDATE);

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 instance, after the assignment

new_string := REPLACE(old_string, NULL, my_string);

the values of old_string and new_string are the same.

If its third argument is null, REPLACE returns its first argument with every occurrence of its second argument removed. For example, after the assignments

syllabified_name := 'Gold-i-locks';
name := REPLACE(syllabified_name, '-', NULL);

the value of name is ‘goldilocks’

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

Built-In Functions

PL/SQL provides many powerful functions to help you manipulate data. These built-in functions fall into the following categories:

- error reporting
- number
- character
- datatype conversion
- date
- object reference
- miscellaneous

Table 2–3 shows the functions in each category. For descriptions of the error-reporting functions, see Chapter 13. For descriptions of the other functions, see Oracle9i SQL Reference.
Except for the error-reporting functions SQLCODE and SQLERRM, you can use all the functions in SQL statements. Also, except for the object-reference functions DEREF, REF, and VALUE and the miscellaneous functions DECODE, DUMP, and VSIZE, you can use all the functions in procedural statements.

Although the SQL aggregate functions (such as AVG and COUNT) and the SQL analytic functions (such as CORR and LAG) are not built into PL/SQL, you can use them in SQL statements (but not in procedural statements).

### Table 2–3  Built-In Functions

<table>
<thead>
<tr>
<th>Error</th>
<th>Number</th>
<th>Character</th>
<th>Conversion</th>
<th>Date</th>
<th>Obj Ref</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCODE</td>
<td>ABS</td>
<td>ASCII</td>
<td>CHARTOROWID</td>
<td>ADD_MONTHS</td>
<td>DEREF</td>
<td>BFILNENAME</td>
</tr>
<tr>
<td>SQLERRM</td>
<td>ACOS</td>
<td>CHR</td>
<td>CONVERT</td>
<td>CURRENT_DATE</td>
<td>REF</td>
<td>DECODE</td>
</tr>
<tr>
<td></td>
<td>ASIN</td>
<td>CONCAT</td>
<td>HEXTORAW</td>
<td>CURRENT_TIMESTAMP</td>
<td>VALUE</td>
<td>DUMP</td>
</tr>
<tr>
<td></td>
<td>ATAN</td>
<td>INITCAP</td>
<td>RAWTOHEX</td>
<td>DBTIMEZONE</td>
<td></td>
<td>EMPTY_BLOB</td>
</tr>
<tr>
<td></td>
<td>ATAN2</td>
<td>INSTR</td>
<td>ROWIDTOCHAR</td>
<td>EXTRACT</td>
<td></td>
<td>EMPTY_CLOB</td>
</tr>
<tr>
<td></td>
<td>BITAND</td>
<td>INSTRB</td>
<td>TO_BLOB</td>
<td>FROM_TZ</td>
<td></td>
<td>GREATEST</td>
</tr>
<tr>
<td></td>
<td>CEIL</td>
<td>LENGTH</td>
<td>TO_CHAR</td>
<td>LAST_DAY</td>
<td></td>
<td>LEAST</td>
</tr>
<tr>
<td></td>
<td>COS</td>
<td>LENGTHB</td>
<td>TO_CLOB</td>
<td>LOCALTIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COSH</td>
<td>LOWER</td>
<td>TO_DATE</td>
<td>MONTHS_BETWEEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXP</td>
<td>LPAD</td>
<td>TO_MULTI_BYTE</td>
<td>NEW_TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOOR</td>
<td>LTRIM</td>
<td>TO_NCHAR</td>
<td>NEXT_DAY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>NLS_INITCAP</td>
<td>TO_NUMBER</td>
<td>NUMTODSINTERVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOG</td>
<td>NLS_LOWER</td>
<td>TO_SINGLE_BYTE</td>
<td>NUMTOYMINTERVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOD</td>
<td>NLS_UPPER</td>
<td></td>
<td>ROUND</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>POWER</td>
<td></td>
<td></td>
<td>RESETTIMEZONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROUND</td>
<td>REPLACE</td>
<td></td>
<td>SYSDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIGN</td>
<td>RPAD</td>
<td></td>
<td>SYSTIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIN</td>
<td>RTRIM</td>
<td></td>
<td>TO_DSINTERVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SINH</td>
<td>SOUNDEX</td>
<td></td>
<td>TO_TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT</td>
<td>SUBSTR</td>
<td></td>
<td>TO_TIMESTAMP_LTZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TAN</td>
<td>SUBSTRB</td>
<td></td>
<td>TO_TIMESTAMP_TZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TANH</td>
<td>TRANSLATE</td>
<td></td>
<td>TO_YMINTERVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRUNC</td>
<td>TRIM</td>
<td></td>
<td>TZ_OFFSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPPER</td>
<td></td>
<td>TRUNC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Like—but oh how different! —William Wordsworth

Every constant, variable, and parameter has a datatype (or type), which specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined datatypes. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and LOB types. In addition, PL/SQL lets you define your own subtypes. This chapter covers the basic types used frequently in PL/SQL programs. Later chapters cover the more specialized types.

This chapter discusses the following topics:

- Predefined Datatypes
- User-Defined Subtypes
- Datatype Conversion
Predefined Datatypes

A scalar type has no internal components. A composite type has internal components that can be manipulated individually. A reference type holds values, called pointers, that designate other program items. A LOB type holds values, called lob locators, that specify the location of large objects (graphic images for example) stored out-of-line.

Figure 3–1 shows the predefined datatypes available for your use. The scalar types fall into four families, which store number, character, Boolean, and date/time data, respectively.

Figure 3–1  Built-in Datatypes

<table>
<thead>
<tr>
<th>Scalar Types</th>
<th>Composite Types</th>
<th>Reference Types</th>
<th>LOB Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY_INTEGER</td>
<td>CHAR</td>
<td>RECORD</td>
<td>BFILE</td>
</tr>
<tr>
<td>DEC</td>
<td>CHARACTER</td>
<td>TABLE</td>
<td>BLOB</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>LONG</td>
<td>VARRAY</td>
<td>CLOB</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>LONG RAW</td>
<td>REF CURSOR</td>
<td>NCLOB</td>
</tr>
<tr>
<td>FLOAT</td>
<td>NCHAR</td>
<td>REF object_type</td>
<td>NVARCHAR2</td>
</tr>
<tr>
<td>INT</td>
<td>NUMBER</td>
<td>RAW</td>
<td>RAW</td>
</tr>
<tr>
<td>INTEGER</td>
<td>NUMERIC</td>
<td>ROWID</td>
<td>STRING</td>
</tr>
<tr>
<td>NATURAL</td>
<td>VARCHAR</td>
<td>STRING</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>NATURALN</td>
<td>NUMBER</td>
<td>UROWID</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>NUMBER</td>
<td>NUMERIC</td>
<td>BOOLEAN</td>
<td>CHAR</td>
</tr>
<tr>
<td>NUMERIC</td>
<td></td>
<td></td>
<td>CHARACTER</td>
</tr>
<tr>
<td>PLS_INTEGER</td>
<td></td>
<td></td>
<td>LONG</td>
</tr>
<tr>
<td>POSITIVE</td>
<td></td>
<td></td>
<td>LONG RAW</td>
</tr>
<tr>
<td>POSITIVEN</td>
<td></td>
<td></td>
<td>NCHAR</td>
</tr>
<tr>
<td>REAL</td>
<td></td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>SIGNTYPE</td>
<td></td>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td></td>
<td>CHAR</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td></td>
<td></td>
<td>DEC</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td></td>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td></td>
<td></td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td></td>
<td></td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

3-2 PL/SQL User’s Guide and Reference
Number Types

Number types let you store numeric data (integers, real numbers, and floating-point numbers), represent quantities, and do calculations.

**BINARY_INTEGER**

You use the `BINARY_INTEGER` datatype to store signed integers. Its magnitude range is $-2^{31} .. 2^{31}$. Like `PLS_INTEGER` values, `BINARY_INTEGER` values require less storage than `NUMBER` values. However, most `BINARY_INTEGER` operations are slower than `PLS_INTEGER` operations.

**BINARY_INTEGER Subtypes**  A *base type* is the datatype from which a subtype is derived. A *subtype* associates a base type with a constraint and so defines a subset of values. For your convenience, PL/SQL predefines the following `BINARY_INTEGER` subtypes:

- **NATURAL**
- **NATURALN**
- **POSITIVE**
- **POSITIVEN**
- **SIGNTYPE**

The subtypes `NATURAL` and `POSITIVE` let you restrict an integer variable to non-negative or positive values, respectively. `NATURALN` and `POSITIVEN` prevent the assigning of nulls to an integer variable. `SIGNTYPE` lets you restrict an integer variable to the values -1, 0, and 1, which is useful in programming tri-state logic.

**NUMBER**

You use the `NUMBER` datatype to store fixed-point or floating-point numbers. Its magnitude range is $1E^{-130} .. 10E^{125}$. If the value of an expression falls outside this range, you get a *numeric overflow or underflow* error. You can specify *precision*, which is the total number of digits, and *scale*, which is the number of digits to the right of the decimal point. The syntax follows:

```plaintext
NUMBER[(precision,scale)]
```

To declare fixed-point numbers, for which you must specify *scale*, use the following form:

```plaintext
NUMBER(precision,scale)
```
To declare floating-point numbers, for which you cannot specify `precision` or `scale` because the decimal point can "float" to any position, use the following form:

```
NUMBER
```

To declare integers, which have no decimal point, use this form:

```
NUMBER(precision) -- same as NUMBER(precision,0)
```

You cannot use constants or variables to specify `precision` and `scale`; you must use integer literals. The maximum precision of a `NUMBER` value is 38 decimal digits. If you do not specify `precision`, it defaults to 38 or the maximum supported by your system, whichever is less.

Scale, which can range from -84 to 127, determines where rounding occurs. For instance, a scale of 2 rounds to the nearest hundredth (3.456 becomes 3.46). A negative scale rounds to the left of the decimal point. For example, a scale of -3 rounds to the nearest thousand (3456 becomes 3000). A scale of 0 rounds to the nearest whole number. If you do not specify `scale`, it defaults to 0.

**NUMBER Subtypes** You can use the following `NUMBER` subtypes for compatibility with ANSI/ISO and IBM types or when you want a more descriptive name:

- `DEC`
- `DECIMAL`
- `DOUBLE PRECISION`
- `FLOAT`
- `INTEGER`
- `INT`
- `NUMERIC`
- `REAL`
- `SMALLINT`

Use the subtypes `DEC`, `DECIMAL`, and `NUMERIC` to declare fixed-point numbers with a maximum precision of 38 decimal digits.

Use the subtypes `DOUBLE PRECISION` and `FLOAT` to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype `REAL` to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent to 18 decimal digits.

Use the subtypes `INTEGER`, `INT`, and `SMALLINT` to declare integers with a maximum precision of 38 decimal digits.
PLS_INTEGER
You use the PLS_INTEGER datatype to store signed integers. Its magnitude range is \(-2^{31} .. 2^{31}\). PLS_INTEGER values require less storage than NUMBER values. Also, PLS_INTEGER operations use machine arithmetic, so they are faster than NUMBER and BINARY_INTEGER operations, which use library arithmetic. For efficiency, use PLS_INTEGER for all calculations that fall within its magnitude range.

Although PLS_INTEGER and BINARY_INTEGER have the same magnitude range, they are not fully compatible. When a PLS_INTEGER calculation overflows, an exception is raised. However, when a BINARY_INTEGER calculation overflows, no exception is raised if the result is assigned to a NUMBER variable.

Because of this small semantic difference, you might want to continue using BINARY_INTEGER in old applications for compatibility. In new applications, always use PLS_INTEGER for better performance.

Character Types
Character types let you store alphanumeric data, represent words and text, and manipulate character strings.

CHAR
You use the CHAR datatype to store fixed-length character data. How the data is represented internally depends on the database character set. The CHAR datatype takes an optional parameter that lets you specify a maximum size up to 32767 bytes. You can specify the size in terms of bytes or characters, where each character contains one or more bytes, depending on the character set encoding. The syntax follows:

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

You cannot use a symbolic constant or variable to specify the maximum size; you must use an integer literal in the range 1 .. 32767.

If you do not specify a maximum size, it defaults to 1. If you specify the maximum size in bytes rather than characters, a CHAR(n) variable might be too small to hold n multibyte characters. To avoid this possibility, use the notation CHAR(n CHAR) so that the variable can hold n characters in the database character set, even if some of those characters contain multiple bytes. When you specify the length in characters, the upper limit is still 32767 bytes. So for double-byte and multibyte character sets, you can only specify 1/2 or 1/3 as many characters as with a single-byte character set.
Although PL/SQL character variables can be relatively long, the maximum width of a CHAR database column is 2000 bytes. So, you cannot insert CHAR values longer than 2000 bytes into a CHAR database column.

You can insert any CHAR(n) value into a LONG database column because the maximum width of a LONG column is 2**31 bytes or two gigabytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG column into a CHAR(n) variable.

When you do not use the CHAR or BYTE qualifiers, the default is determined by the setting of the NLS_LENGTH_SEMANTICS initialization parameter. When a PL/SQL procedure is compiled, the setting of this parameter is recorded, so that the same setting is used when the procedure is recompiled after being invalidated.

**Note:** Semantic differences between the CHAR and VARCHAR2 base types are discussed in Appendix B.

**CHAR Subtype** The CHAR subtype CHARACTER has the same range of values as its base type. That is, CHARACTER is just another name for CHAR. You can use this subtype for compatibility with ANSI/ISO and IBM types or when you want an identifier more descriptive than CHAR.

**LONG and LONG RAW**

You use the LONG datatype to store variable-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum size of a LONG value is 32760 bytes.

You use the LONG RAW datatype to store binary data 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 32760 bytes.

Starting in Oracle9i, LOB variables can be used interchangeably with LONG and LONG RAW variables. Oracle recommends migrating any LONG data to the CLOB type, and any LONG RAW data to the BLOB type. See "LOB Types" on page 3-13 for more details.

You can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2**31 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG column into a LONG variable.

Likewise, you can insert any LONG RAW value into a LONG RAW database column because the maximum width of a LONG RAW column is 2**31 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG RAW column into a LONG RAW variable.
LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, SQL function calls, or certain SQL clauses such as WHERE, GROUP BY, and CONNECT BY. For more information, see Oracle9i SQL Reference.

Note: In SQL statements, PL/SQL binds LONG values as VARCHAR2, not as LONG. However, if the length of the bound VARCHAR2 exceeds the maximum width of a VARCHAR2 column (4000 bytes), Oracle converts the bind type to LONG automatically, then issues an error message because you cannot pass LONG values to a SQL function.

RAW

You use the RAW datatype to store binary data or byte strings. For example, a RAW variable might store a sequence of graphics characters or a digitized picture. Raw data is like VARCHAR2 data, except that PL/SQL does not interpret raw data. Likewise, Oracle Net does no character set conversions when you transmit raw data from one system to another.

The RAW datatype takes a required parameter that lets you specify a maximum size up to 32767 bytes. The syntax follows:

RAW(maximum_size)

You cannot use a symbolic constant or variable to specify the maximum size; you must use an integer literal in the range 1 .. 32767.

The maximum width of a RAW database column is 2000 bytes. So, you cannot insert RAW values longer than 2000 bytes into a RAW column. You can insert any RAW value into a LONG RAW database column because the maximum width of a LONG RAW column is 2**31 bytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG RAW column into a RAW variable.

ROWID and UROWID

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

Suggestion: Use the ROWID datatype only for backward compatibility with old applications. For new applications, use the UROWID datatype.
When you select or fetch a rowid into a \texttt{ROWID} variable, you can use the built-in function \texttt{ROWIDTOCHAR}, which converts the binary value into an 18-byte character string. Conversely, the function \texttt{CHARTROWID} converts a \texttt{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 \texttt{SYS_INVALID_ROWID}. This also applies to implicit conversions.

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

\textbf{Physical Rowids} Physical rowids provide fast access to particular rows. As long as the row exists, its physical rowid does not change. Efficient and stable, physical rowids are useful for selecting a set of rows, operating on the whole set, and then updating a subset. For example, you can compare a \texttt{UROWID} variable with the \texttt{ROWID} pseudocolumn in the \texttt{WHERE} clause of an \texttt{UPDATE} or \texttt{DELETE} statement to identify the latest row fetched from a cursor. See "Fetching Across Commits" on page 6-51.

A physical rowid can have either of two formats. The 10-byte \textit{extended rowid format} supports tablespace-relative block addresses and can identify rows in partitioned and non-partitioned tables. The 6-byte \textit{restricted rowid format} is provided for backward compatibility.

Extended rowids use a base-64 encoding of the physical address for each row selected. For example, in SQL*Plus (which implicitly converts rowids into character strings), the query

\begin{verbatim}
SQL> SELECT rowid, ename FROM emp WHERE empno = 7788;
\end{verbatim}

might return the following row:

\begin{verbatim}
ROWID              ENAME
------------------ ----------
AAAAqcAABAAADFNAAH SCOTT
\end{verbatim}

The format, OOOOOOFFBBBBBRRR, has four parts:

- \textbf{OOOOOO}: The data object number (AAAAqc in the example above) identifies the database segment. Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- \textbf{FFF}: The file number (AAB in the example) identifies the data file that contains the row. File numbers are unique within a database.
Predefined Datatypes

- **BBBBBB**: The block number (AAADFN in the example) identifies the data block that contains the row. Block numbers are relative to their data file, not their tablespace. So, two rows in the same tablespace but in different data files can have the same block number.

- **RRR**: The row number (AAH in the example) identifies the row in the block.

**Logical Rowids** Logical rowids provide the fastest access to particular rows. Oracle uses them to construct secondary indexes on index-organized tables. Having no permanent physical address, a logical rowid can move across data blocks when new rows are inserted. However, if the physical location of a row changes, its logical rowid remains valid.

A logical rowid can include a guess, which identifies the block location of a row at the time the guess is made. Instead of doing a full key search, Oracle uses the guess to search the block directly. However, as new rows are inserted, guesses can become stale and slow down access to rows. To obtain fresh guesses, you can rebuild the secondary index.

You can use the ROWID pseudocolumn to select logical rowids (which are opaque values) from an index-organized table. Also, you can insert logical rowids into a column of type UROWID, which has a maximum size of 4000 bytes.

The ANALYZE statement helps you track the staleness of guesses. This is useful for applications that store rowids with guesses in a UROWID column, then use the rowids to fetch rows.

**Note**: To manipulate rowids, you can use the supplied package DBMS_ROWID. For more information, see Oracle9i Supplied PL/SQL Packages and Types Reference.

**VARCHAR2**

You use the VARCHAR2 datatype to store variable-length character data. How the data is represented internally depends on the database character set. The VARCHAR2 datatype takes a required parameter that specifies a maximum size up to 32767 bytes. The syntax follows:

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

You cannot use a symbolic constant or variable to specify the maximum size; you must use an integer literal in the range 1 .. 32767.

Small VARCHAR2 variables are optimized for performance, and larger ones are optimized for efficient memory use. The cutoff point is 2000 bytes. For a VARCHAR2 that is 2000 bytes or longer, PL/SQL dynamically allocates only enough memory to...
hold the actual value. For a VARCHAR2 variable that is shorter than 2000 bytes, PL/SQL preallocates the full declared length of the variable. For example, if you assign the same 500-byte value to a VARCHAR2 (2000 BYTE) variable and to a VARCHAR2 (1999 BYTE) variable, the former takes up 500 bytes and the latter takes up 1999 bytes.

If you specify the maximum size in bytes rather than characters, a VARCHAR2 (n) variable might be too small to hold n multibyte characters. To avoid this possibility, use the notation VARCHAR2 (n CHAR) so that the variable can hold n characters in the database character set, even if some of those characters contain multiple bytes. When you specify the length in characters, the upper limit is still 32767 bytes. So for double-byte and multibyte character sets, you can only specify 1/2 or 1/3 as many characters as with a single-byte character set.

Although PL/SQL character variables can be relatively long, the maximum width of a VARCHAR2 database column is 4000 bytes. So, you cannot insert VARCHAR2 values longer than 4000 bytes into a VARCHAR2 database column.

You can insert any VARCHAR2 (n) value into a LONG database column because the maximum width of a LONG column is 2**31 bytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG column into a VARCHAR2 (n) variable.

When you do not use the CHAR or BYTE qualifiers, the default is determined by the setting of the NLS_LENGTH_SEMANTICS initialization parameter. When a PL/SQL procedure is compiled, the setting of this parameter is recorded, so that the same setting is used when the procedure is recompiled after being invalidated.

VARCHAR2 Subtypes  The VARCHAR2 subtypes below have the same range of values as their base type. For example, VARCHAR is just another name for VARCHAR2.

STRING
 VARCHAR

You can use these subtypes for compatibility with ANSI/ISO and IBM types.

Note: Currently, VARCHAR is synonymous with VARCHAR2. However, in future releases of PL/SQL, to accommodate emerging SQL standards, VARCHAR might become a separate datatype with different comparison semantics. So, it is a good idea to use VARCHAR2 rather than VARCHAR.

National Character Types

The widely used one-byte ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, but some Asian languages, such as Japanese, contain
Predefined Datatypes

thousands of characters. These languages require two or three bytes to represent each character. To deal with such languages, Oracle provides globalization support, which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments.

With globalization support, number and date formats adapt automatically to the language conventions specified for a user session. Thus, users around the world can interact with Oracle in their native languages.

PL/SQL supports two character sets called the database character set, which is used for identifiers and source code, and the national character set, which is used for national language data. The datatypes NCHAR and NVARCHAR2 store character strings formed from the national character set.

Note: When converting CHAR or VARCHAR2 data between databases with different character sets, make sure the data consists of well-formed strings. For more information, see Oracle9i Globalization and National Language Support Guide.

Comparing UTF8 and AL16UTF16 Encodings

The national character set represents data as Unicode, using either the UTF8 or AL16UTF16 encoding.

Each character in the AL16UTF16 encoding takes up 2 bytes. This makes it simple to calculate string lengths to avoid truncation errors when mixing different programming languages, but requires extra storage overhead to store strings made up mostly of ASCII characters.

Each character in the UTF8 encoding takes up 1, 2, or 3 bytes. This lets you fit more characters into a variable or table column, but only if most characters can be represented in a single byte. It introduces the possibility of truncation errors when transferring the data to a buffer measured in bytes.

Oracle Corporation recommends that you use the default AL16UTF16 encoding wherever practical, for maximum runtime reliability. If you need to determine how many bytes are required to hold a Unicode string, use the LENGTHB function rather than LENGTH.

NCHAR

You use the NCHAR datatype to store fixed-length (blank-padded if necessary) national character data. How the data is represented internally depends on the national character set specified when the database was created, which might use a variable-width encoding (UTF8) or a fixed-width encoding (AL16UTF16). Because
this type can always accommodate multibyte characters, you can use it to hold any Unicode character data.

The NCHAR datatype takes an optional parameter that lets you specify a maximum size in characters. The syntax follows:

\[ \text{NCHAR}\left(\text{maximum}_\text{size}\right) \]

Because the physical limit is 32767 bytes, the maximum value you can specify for the length is \( \frac{32767}{2} \) in the \text{AL16UTF16} encoding, and \( \frac{32767}{3} \) in the \text{UTF8} encoding.

You cannot use a symbolic constant or variable to specify the maximum size; you must use an integer literal.

If you do not specify a maximum size, it defaults to 1. The value always represents the number of characters, unlike \text{CHAR} which can be specified in either characters or bytes.

\[
\text{my\_string NCHAR(100); \quad -- maximum size is 100 characters}
\]

The maximum width of an NCHAR database column is 2000 bytes. So, you cannot insert NCHAR values longer than 2000 bytes into an NCHAR column.

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

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

\textit{NVARCHAR2}

You use the NVARCHAR2 datatype to store variable-length Unicode character data. How the data is represented internally depends on the national character set specified when the database was created, which might use a variable-width encoding (UTF8) or a fixed-width encoding (AL16UTF16). Because this type can always accommodate multibyte characters, you can use it to hold any Unicode character data.

The NVARCHAR2 datatype takes a required parameter that specifies a maximum size in characters. The syntax follows:

\[ \text{NVARCHAR2}\left(\text{maximum}_\text{size}\right) \]
Because the physical limit is 32767 bytes, the maximum value you can specify for the length is 32767/2 in the AL16UTF16 encoding, and 32767/3 in the UTF8 encoding.

You cannot use a symbolic constant or variable to specify the maximum size; you must use an integer literal.

The maximum size always represents the number of characters, unlike VARCHAR2 which can be specified in either characters or bytes.

my_string NVARCHAR2(200);  -- maximum size is 200 characters

The maximum width of a NVARCHAR2 database column is 4000 bytes. Therefore, you cannot insert NVARCHAR2 values longer than 4000 bytes into a NVARCHAR2 column.

You can interchange VARCHAR2 and NVARCHAR2 values in statements and expressions. It is always safe to turn a VARCHAR2 value into an NVARCHAR2 value, but turning an NVARCHAR2 value into 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 can result in characters that usually look like question marks (?).

**LOB Types**

The LOB (large object) datatypes BFILE, BLOB, CLOB, and NCLOB let you store blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) up to four gigabytes in size. And, they allow efficient, random, piece-wise access to the data.

The LOB types differ from the LONG and LONG RAW types in several ways. For example, LOBs (except NCLOB) can be attributes of an object type, but LONGs cannot. The maximum size of a LOB is four gigabytes, but the maximum size of a LONG is two gigabytes. Also, LOBs support random access to data, but LONGs support only sequential access.

LOB types store *lob locators*, which point to large objects stored in an external file, *in-line* (inside the row) or *out-of-line* (outside the row). Database columns of type BLOB, CLOB, NCLOB, or BFILE store the locators. BLOB, CLOB, and NCLOB data is stored in the database, in or outside the row. BFILE data is stored in operating system files outside the database.

PL/SQL operates on LOBs through the locators. For example, when you select a BLOB column value, only a locator is returned. If you got it during a transaction, the LOB locator includes a transaction ID, so you cannot use it to update the LOB in
another transaction. Likewise, you cannot save a LOB locator during one session, then use it in another session.

Starting in Oracle9i, you can also convert CLOBs to CHAR and VARCHAR2 types and vice versa, or BLOBs to RAW and vice versa, which lets you use LOB types in most SQL and PL/SQL statements and functions. To read, write, and do piecewise operations on LOBs, you can use the supplied package DBMS_LOB. For more information, see Oracle9i Application Developer’s Guide - Large Objects (LOBs).

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

BFILEs are read-only, so you cannot modify them. The size of a BFILE is system dependent but cannot exceed four gigabytes (2**32 - 1 bytes). 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
You use the BLOB datatype to store large binary objects in the database, in-line or out-of-line. Every BLOB variable stores a locator, which points to a large binary object. The size of a BLOB cannot exceed four gigabytes.

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
You use the CLOB datatype to store large blocks of character data in the database, in-line or out-of-line. Both fixed-width and variable-width character sets are supported. Every CLOB variable stores a locator, which points to a large block of character data. The size of a CLOB cannot exceed four gigabytes.

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
You use the NCLOB datatype to store large blocks of NCHAR data in the database, in-line or out-of-line. Both fixed-width and variable-width character sets are supported. Every NCLOB variable stores a locator, which points to a large block of NCHAR data. The size of an NCLOB cannot exceed four gigabytes.

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.

Boolean Type

BOOLEAN
You use the BOOLEAN datatype to store the logical values TRUE, FALSE, and NULL (which stands for a missing, unknown, or inapplicable value). Only logic operations are allowed on BOOLEAN variables.

The BOOLEAN datatype takes no parameters. Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable. You cannot insert the values TRUE and FALSE into a database column. Also, you cannot select or fetch column values into a BOOLEAN variable.

Datetime and Interval Types

The datatypes in this section let you store and manipulate dates, times, and intervals (periods of time). A variable that has a date/time datatype holds values called datetimes; a variable that has an interval datatype holds 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>
</tbody>
</table>
Predefined Datatypes

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 Oracle9i SQL Reference.

**DATE**

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

**Tip:** 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.

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. For example, the following statement returns the number of days since an employee was hired:

```
SELECT SYSDATE - hiredate INTO days_worked FROM emp
  WHERE empno = 7499;
```

In arithmetic expressions, PL/SQL interprets integer literals as days. For instance, `SYSDATE + 1` is tomorrow.

**TIMESTAMP**

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

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

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

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

In the following example, you declare a variable of type `TIMESTAMP`, then assign a literal value to it:

```
DECLARE
  checkout TIMESTAMP(3);
BEGIN
  ...
END;
```

In this example, the fractional part of the seconds field is 0.275.

**TIMESTAMP WITH TIME ZONE**

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

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

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

In the following example, you declare a variable of type TIMESTAMP WITH TIME ZONE, then assign a literal value to it:

```sql
DECLARE
    logoff TIMESTAMP(3) WITH TIME ZONE;
BEGIN
    logoff := '1999-10-31 09:42:37.114 +02:00';
    ...
END;
```

In this example, the time-zone displacement is +02:00.

You can also specify the time zone by using a symbolic name. The specification can include a long form such as 'US/Pacific', an abbreviation such as 'PDT', or a combination. For example, the following literals all represent the same time. The third form is most reliable because it specifies the rules to follow at the point when switching to daylight savings time.

```sql
TIMESTAMP '1999-04-15 8:00:00 -8:00'
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
TIMESTAMP '1999-10-31 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 V$TIMEZONE_NAMES data dictionary view.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of their time-zone displacements. For example, the following two values are considered identical because, in UTC, 8:00 AM Pacific Standard Time is the same as 11:00 AM Eastern Standard Time:

```sql
'1999-08-29 08:00:00 -8:00'
'1999-08-29 11:00:00 -5:00'
```

**TIMESTAMP WITH LOCAL TIME ZONE**

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

The syntax is

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

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

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

In the following example, you declare a variable of type `TIMESTAMP WITH LOCAL TIME ZONE`:

```sql
DECLARE
    logoff TIMESTAMP(3) WITH LOCAL TIME ZONE;
BEGIN
    ...
END;
```

You cannot assign literal values to a variable of this type.

**INTERVAL YEAR TO MONTH**

You use the datatype `INTERVAL YEAR TO MONTH` to store and manipulate intervals of years and months. The syntax is:

```
INTERVAL YEAR[(precision)] TO MONTH
```

where `precision` specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 .. 4. The default is 2.

In the following example, you declare a variable of type `INTERVAL YEAR TO MONTH`, then assign a value of 101 years and 3 months to it:

```sql
DECLARE
    lifetime INTERVAL YEAR(3) TO MONTH;
BEGIN
    lifetime := INTERVAL '101-3' YEAR TO MONTH; -- interval literal
    lifetime := '101-3'; -- implicit conversion from character type
```

You cannot assign literal values to a variable of this type.
 lifet ime := INTERVAL ‘101’ YEAR; -- Can specify just the years
life time := INTERVAL ‘3’ MONTH; -- Can specify just the months
 ... 
END;

INTERVAL DAY TO SECOND

You use the datatype INTERVAL DAY TO SECOND to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is:

INTERVAL DAY[(leading_precision)] TO
SECOND[(fractional_seconds_precision)]

where leading_precision and fractional_seconds_precision specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 .. 9. The defaults are 2 and 6, respectively.

In the following example, you declare a variable of type INTERVAL DAY TO SECOND:

DECLARE
  lag_time INTERVAL DAY(3) TO SECOND(3);
BEGIN
  IF lag_time > INTERVAL ‘6’ DAY THEN ... 
  ... 
END;

Datetime and Interval Arithmetic

PL/SQL lets you construct datetime and interval expressions. The following list shows the operators that you can use in such expressions:

<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>
</tbody>
</table>
You can also manipulate datetime value using various functions, such as `EXTRACT`. For a list of such functions, see Figure 2–3, "Built-In Functions" on page 2-36.

For further information and examples of datetime arithmetic, see Oracle9i SQL Reference and Oracle9i Application Developer's Guide - Fundamentals.

### 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 procedure parameters of these types, you can declare variables and procedure parameters of the following subtypes, which use the maximum values for precision:

- `TIMESTAMP_UNCONSTRAINED`
- `TIMESTAMP_TZ_UNCONSTRAINED`
- `TIMESTAMP_LTZ_UNCONSTRAINED`
- `YMINTERVAL_UNCONSTRAINED`
- `DSINTERVAL_UNCONSTRAINED`

### User-Defined Subtypes

Each PL/SQL base type specifies a set of values and a set of operations applicable to items of that type. Subtypes specify the same set of operations as their base type but only a subset of its values. Thus, a subtype does not introduce a new type; it merely places an optional constraint on its base type.

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:

```plsql
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.

### Defining Subtypes

You can define your own subtypes in the declarative part of any PL/SQL block, subprogram, or package using the syntax

```
SUBTYPE subtype_name IS base_type[(constraint)] [NOT NULL];
```

where `subtype_name` is a type specifier used in subsequent declarations, `base_type` is any scalar or user-defined PL/SQL datatype, and `constraint` applies only to base types that can specify precision and scale or a maximum size.

Some examples follow:

```sql
DECLARE
    SUBTYPE BirthDate IS DATE NOT NULL;  -- based on DATE type
    SUBTYPE Counter IS NATURAL;          -- based on NATURAL subtype
    TYPE NameList IS TABLE OF VARCHAR2(10);
    SUBTYPE DutyRoster IS NameList;      -- based on TABLE type
    TYPE TimeRec IS RECORD (minutes INTEGER, hours INTEGER);
    SUBTYPE FinishTime IS TimeRec;       -- based on RECORD type
    SUBTYPE ID_Num IS emp.empno%TYPE;    -- based on column type
```

You can use `%TYPE` or `%ROWTYPE` to specify the base type. When `%TYPE` provides the datatype of a database column, the subtype inherits the size constraint (if any) of the column. However, the subtype does not inherit other kinds of constraints such as NOT NULL.

### Using Subtypes

Once you define a subtype, you can declare items of that type. In the example below, you declare a variable of type `Counter`. Notice how the subtype name indicates the intended use of the variable.

```sql
DECLARE
    SUBTYPE Counter IS NATURAL;
    rows Counter;
```

The following example shows that you can constrain a user-defined subtype when declaring variables of that type:

```sql
DECLARE
```

3-22 PL/SQL User’s Guide and Reference
SUBTYPE Accumulator IS NUMBER;
total Accumulator(7,2);

Subtypes can increase reliability by detecting out-of-range values. In the example below, you restrict the subtype Numeral to storing integers in the range -9..9. If your program tries to store a number outside that range in a Numeral variable, PL/SQL raises an exception.

DECLARE
  SUBTYPE Numeral IS NUMBER(1,0);
  x_axis Numeral;  -- magnitude range is -9 .. 9
  y_axis Numeral;
BEGIN
  x_axis := 10;  -- raises VALUE_ERROR
  ...
END;

Type Compatibility
An unconstrained subtype is interchangeable with its base type. For example, given the following declarations, the value of amount can be assigned to total without conversion:

DECLARE
  SUBTYPE Accumulator IS NUMBER;
  amount NUMBER(7,2);
  total  Accumulator;
BEGIN
  ...
  total := amount;
  ...
END;

Different subtypes are interchangeable if they have the same base type. For instance, given the following declarations, the value of finished can be assigned to debugging:

DECLARE
  SUBTYPE Sentinel IS BOOLEAN;
  SUBTYPE Switch IS BOOLEAN;
  finished  Sentinel;
  debugging Switch;
BEGIN
  ...
  debugging := finished;
  ...

Datatype Conversion

Different subtypes are also interchangeable if their base types are in the same datatype family. For example, given the following declarations, the value of verb can be assigned to sentence:

```
DECLARE
    SUBTYPE Word IS CHAR(15);
    SUBTYPE Text IS VARCHAR2(1500);
    verb     Word;
    sentence Text(150);
BEGIN
    ...
    sentence := verb;
    ...
END;
```

Datatype Conversion

Sometimes it is necessary to convert a value from one datatype to another. For example, if you want to examine a rowid, you must convert it to a character string. PL/SQL supports both explicit and implicit (automatic) datatype conversion.

Explicit Conversion

To convert values from one datatype to another, you use built-in functions. 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. For more information about these functions, see Oracle9i SQL Reference.

Implicit Conversion

When it makes sense, PL/SQL can convert the datatype of a value implicitly. This lets you use literals, variables, and parameters of one type where another type is expected. In the example below, the CHAR variables start_time and finish_time hold string values representing the number of seconds past midnight. The difference between those values must be assigned to the NUMBER variable elapsed_time. So, PL/SQL converts the CHAR values to NUMBER values automatically.

```
DECLARE
    start_time   CHAR(5);
```
finish_time  CHAR(5);
elapsed_time  NUMBER(5);
BEGIN
  /* Get system time as seconds past midnight. */
  SELECT TO_CHAR(SYSDATE,'SSSSS') INTO start_time FROM sys.dual;
  -- do something
  /* Get system time again. */
  SELECT TO_CHAR(SYSDATE,'SSSSS') INTO finish_time FROM sys.dual;
  /* Compute elapsed time in seconds. */
  elapsed_time := finish_time - start_time;
  INSERT INTO results VALUES (elapsed_time, ...);
END;

Before assigning a selected column value to a variable, PL/SQL will, if necessary, convert the value from the datatype of the source column to the datatype of the variable. This happens, for example, when you select a DATE column value into a VARCHAR2 variable.

Likewise, before assigning the value of a variable to a database column, PL/SQL will, if necessary, convert the value from the datatype of the variable to the datatype of the target column. If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. In such cases, you must use a datatype conversion function. Table 3–1 shows which implicit conversions PL/SQL can do.

Notes:

- The table lists only types that have different representations. Types that have the same representation, such as CLOB and NCLOB, CHAR and NCHAR, and VARCHAR and NVARCHAR2, can be substituted for each other.

- To convert between CLOB and NCLOB, you must use the conversion functions TO_CLOB and TO_NCLOB.

- TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL DAY TO SECOND, and INTERVAL YEAR TO MONTH can all be converted using the same rules as the DATE type. However, because of their different internal representations, these types cannot always be converted to each other. See Oracle9i SQL Reference for details on implicit conversions between different date and time types.
It is your responsibility to ensure that values are convertible. For instance, PL/SQL can convert the \texttt{CHAR} value \texttt{’02–JUN–92’} to a \texttt{DATE} value but cannot convert the \texttt{CHAR} value \texttt{’YESTERDAY’} to a \texttt{DATE} value. Similarly, PL/SQL cannot convert a \texttt{VARCHAR2} value containing alphabetic characters to a \texttt{NUMBER} value.

**Implicit versus Explicit Conversion**

Generally, to rely on implicit datatype conversions is a poor programming practice because they can hamper performance and might change from one software release to the next. Also, implicit conversions are context sensitive and therefore not always predictable. Instead, use datatype conversion functions. That way, your applications will be more reliable and easier to maintain.

**DATE Values**

When you select a \texttt{DATE} column value into a \texttt{CHAR} or \texttt{VARCHAR2} variable, PL/SQL must convert the internal binary value to a character value. So, PL/SQL calls the function \texttt{TO_CHAR}, which returns a character string in the default date format. To get other information such as the time or Julian date, you must call \texttt{TO_CHAR} with a format mask.

A conversion is also necessary when you insert a \texttt{CHAR} or \texttt{VARCHAR2} value into a \texttt{DATE} column. So, PL/SQL calls the function \texttt{TO_DATE}, which expects the default format.
date format. To insert dates in other formats, you must call `TO_DATE` with a format mask.

**RAW and LONG RAW Values**

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. In this case, PL/SQL returns 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.

A 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. If either character does not represent the hexadecimal equivalent of a nibble, PL/SQL raises an exception.
This chapter shows you how to structure the flow of control through a PL/SQL program. You learn how statements are connected by simple but powerful control structures that have a single entry and exit point. Collectively, these structures can handle any situation. Their proper use leads naturally to a well-structured program.

This chapter discusses the following topics:

- Overview of PL/SQL Control Structures
- Conditional Control: IF and CASE Statements
- Iterative Control: LOOP and EXIT Statements
- Sequential Control: GOTO and NULL Statements
Overview of PL/SQL Control Structures

According to the structure theorem, any computer program can be written using the basic control structures shown in Figure 4–1. They can be combined in any way necessary to deal with a given problem.

Figure 4–1  Control Structures

The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Conditional Control: IF and CASE Statements

Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions.
Conditional Control: IF and CASE Statements

**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 follows:

```
IF condition THEN
    sequence_of_statements
END IF;
```

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. An example follows:

```
IF sales > quota THEN
    compute_bonus(empid);
    UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

You might want to place brief IF statements on a single line, as in

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

**IF-THEN-ELSE Statement**

The second form of IF statement adds the keyword **ELSE** followed by an alternative sequence of statements, as follows:

```
IF condition THEN
    sequence_of_statements1
ELSE
    sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = ’CR’ THEN
    UPDATE accounts SET balance = balance + credit WHERE ... 
ELSE
    UPDATE accounts SET balance = balance - debit WHERE ... 
END IF;
```
The **THEN** and **ELSE** clauses can include **IF** statements. That is, **IF** statements can be nested, as the following example shows:

```sql
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit WHERE ... 
ELSE
  IF new_balance >= minimum_balance THEN
    UPDATE accounts SET balance = balance - debit WHERE ...
  ELSE
    RAISE insufficient_funds;
  END IF;
END IF;
```

**IF-THEN-ELSIF Statement**

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of **IF** statement uses the keyword **ELSIF** (not **ELSEIF**) to introduce additional conditions, as follows:

```sql
IF condition1 THEN
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

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. Consider the following example:

```sql
BEGIN
  ... 
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```
If the value of sales is larger than 50000, the first and second conditions are true. Nevertheless, bonus is assigned the proper value of 1500 because the second condition is never tested. When the first condition is true, its associated statement is executed and control passes to the INSERT statement.

**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. (Recall from Chapter 2 that a selector is an expression whose value is used to select one of several alternatives.) To compare the IF and CASE statements, consider the following code that outputs descriptions of school grades:

```plsql
IF grade = 'A' THEN
    dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
    dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
    dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
    dbms_output.put_line('Fair');
ELSIF grade = 'F' THEN
    dbms_output.put_line('Poor');
ELSE
    dbms_output.put_line('No such grade');
END IF;
```

Notice the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. Let us rewrite the preceding code using the CASE statement, as follows:

```plsql
CASE grade
    WHEN 'A' THEN dbms_output.put_line('Excellent');
    WHEN 'B' THEN dbms_output.put_line('Very Good');
    WHEN 'C' THEN dbms_output.put_line('Good');
    WHEN 'D' THEN dbms_output.put_line('Fair');
    WHEN 'F' THEN dbms_output.put_line('Poor');
    ELSE dbms_output.put_line('No such grade');
END CASE;
```

The CASE statement is more readable and more efficient. So, when possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.
The `CASE` statement begins with the keyword `CASE`. The keyword is followed by a selector, which is the variable `grade` in the last example. The selector expression can be arbitrarily complex. For example, it can contain function calls. Usually, however, it consists of a single variable. The selector expression is evaluated only once. The value it yields can have any PL/SQL datatype other than `BLOB`, `BFILE`, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

The selector is followed by one or more `WHEN` clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a `WHEN`-clause expression, that `WHEN` clause is executed. For instance, 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;
```

If the `CASE` statement selects the implicit `ELSE` clause, PL/SQL raises the predefined exception `CASE_NOT_FOUND`. So, there is always a default action, even when you omit the `ELSE` clause.

The keywords `END CASE` terminate the `CASE` statement. These two keywords must be separated by a space. The `CASE` statement has the following form:

```
[<<label_name>>]
CASE selector
    WHEN expression1 THEN sequence_of_statements1;
    WHEN expression2 THEN sequence_of_statements2;
    ...
    WHEN expressionN THEN sequence_of_statementsN;
[ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

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 \texttt{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 \texttt{CASE} statement is the \texttt{CASE} expression, where each \texttt{WHEN} clause is an expression. For details, see "CASE Expressions" on page 2-31.

\textbf{Searched CASE Statement}

PL/SQL also provides a \textit{searched} \texttt{CASE} statement, which has the form:

\begin{verbatim}
[<<label_name>>]
CASE
  WHEN search\_condition1 THEN sequence\_of\_statements1;
  WHEN search\_condition2 THEN sequence\_of\_statements2;
  ...
  WHEN search\_conditionN THEN sequence\_of\_statementsN;
[ELSE sequence\_of\_statementsN+1;
END CASE [label_name];
\end{verbatim}

The searched \texttt{CASE} statement has no selector. Also, its \texttt{WHEN} clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type. An example follows:

\begin{verbatim}
CASE
  WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
  WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
  WHEN grade = 'C' THEN dbms_output.put_line('Good');
  WHEN grade = 'D' THEN dbms_output.put_line('Fair');
  WHEN grade = 'F' THEN dbms_output.put_line('Poor');
  ELSE dbms_output.put_line('No such grade');
END CASE;
\end{verbatim}

The search conditions are evaluated sequentially. The Boolean value of each search condition determines which \texttt{WHEN} clause is executed. If a search condition yields \texttt{TRUE}, its \texttt{WHEN} clause is executed. If any \texttt{WHEN} clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.

If none of the search conditions yields \texttt{TRUE}, the \texttt{ELSE} clause is executed. The \texttt{ELSE} clause is optional. However, if you omit the \texttt{ELSE} clause, PL/SQL adds the following implicit \texttt{ELSE} clause:

\begin{verbatim}
ELSE RAISE CASE\_NOT\_FOUND;
\end{verbatim}
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 PL/SQL Conditional Statements

Avoid clumsy IF statements like those in the following example:

```sql
IF new_balance < minimum_balance THEN
  overdrawn := TRUE;
ELSE
  overdrawn := FALSE;
END IF;
...
IF overdrawn = TRUE THEN
  RAISE insufficient_funds;
END IF;
```

This code disregards two useful facts. First, the value of a Boolean expression can be assigned directly to a Boolean variable. So, you can replace the first IF statement with a simple assignment, as follows:

```sql
overdrawn := new_balance < minimum_balance;
```

Second, a Boolean variable is itself either true or false. So, you can simplify the condition in the second IF statement, as follows:

```sql
IF overdrawn THEN ...
```

When possible, use the ELSIF clause instead of nested IF statements. That way, your code will be easier to read and understand. Compare the following IF statements:

```sql
IF condition1 THEN
  statement1;
ELSE
  IF condition2 THEN
    statement2;
  ELSE
    IF condition3 THEN
      statement3;
    END IF;
  END IF;
END IF;
```

```sql
IF condition1 THEN
  statement1;
ELSIF condition2 THEN
  statement2;
ELSIF condition3 THEN
  statement3;
END IF;
```
These statements are logically equivalent, but the first statement obscures the flow of logic, whereas the second statement reveals it.

If you are comparing 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.

**Iterative Control: LOOP and EXIT Statements**

**LOOP** statements let you execute a sequence of statements multiple times. There are three forms of **LOOP** statements: **LOOP**, **WHILE-LOOP**, and **FOR-LOOP**.

The simplest form of **LOOP** statement is the basic (or infinite) 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. If further processing is undesirable or impossible, you can use an **EXIT** statement to complete the loop. You can place one or more **EXIT** statements anywhere inside a loop, but nowhere outside a loop. There are two forms of **EXIT** statements: **EXIT** and **EXIT-WHEN**.

**EXIT**

The **EXIT** statement forces a loop to complete unconditionally. When an **EXIT** statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
  ...
  IF credit_rating < 3 THEN
    ...
    EXIT;  -- exit loop immediately
  END IF;
END LOOP;
-- control resumes here
```

The next example shows that you cannot use the **EXIT** statement to complete a **PL/SQL** block:
BEGIN
  ...
  IF credit_rating < 3 THEN
    ...
    EXIT; -- illegal
  END IF;
END;

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement. For more information, see "Using the RETURN Statement" on page 8-8.

EXIT-WHEN
The EXIT-WHEN statement lets a loop complete conditionally. 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 after the loop. An example follows:

LOOP
  FETCH c1 INTO ...  
  EXIT WHEN c1%NOTFOUND; -- exit loop if condition is true
  ...
END LOOP;
CLOSE c1;

Until the condition is true, the loop cannot complete. So, a statement inside the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.

The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:

IF count > 100 THEN   |   EXIT WHEN count > 100;
EXIT;               |   END IF;

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.
Loop Labels
Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```sql
<<label_name>>
LOOP
  sequence_of_statements
END LOOP;
```

Optionally, the label name can also appear at the end of the LOOP statement, as the following example shows:

```sql
<<my_loop>>
LOOP
  ...
END LOOP my_loop;
```

When you nest labeled loops, use ending label names to improve readability. With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement, as follows:

```sql
<<outer>>
LOOP
  ...
  LOOP
    ...
    EXIT outer WHEN ... -- exit both loops
    END LOOP;
  ...
END LOOP outer;
```

Every enclosing loop up to and including the labeled loop is exited.

WHILE-LOOP
The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
```
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. An example follows:

```
WHILE total <= 25000 LOOP
    ...
    SELECT sal INTO salary FROM emp WHERE ...
    total := total + salary;
END LOOP;
```

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. In the last example, if the initial value of `total` is larger than 25000, the condition is false and the loop is bypassed.

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. Therefore, the sequence of statements is executed at least once. PL/SQL has no such structure, but you can easily build one, as follows:

```
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. Otherwise, you have an infinite loop. For example, the following `LOOP` statements are logically equivalent:

```
WHILE TRUE LOOP     |     LOOP
    ...
   END LOOP;           |     END LOOP;
```

FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR 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. A double dot (..) serves as the range operator. The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated.

As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.

```
FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
    sequence_of_statements  -- executes three times
END LOOP;
```

The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:

```
FOR i IN 3..3 LOOP  -- assign the value 3 to i
    sequence_of_statements  -- executes one time
END LOOP;
```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, 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. Nevertheless, you write the range bounds in ascending (not descending) order.

```
FOR i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
    sequence_of_statements  -- executes three times
END LOOP;
```

Inside a FOR loop, the loop counter can be referenced like a constant but cannot be assigned values, as the following example shows:

```
FOR ctr IN 1..10 LOOP
    IF NOT finished THEN
        INSERT INTO ... VALUES (ctr, ...);  -- legal
        factor := ctr * 2;  -- legal
    ELSE
```

```
Iteration Schemes

The bounds of a loop range can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR. The lower bound need not be 1, as the examples below show. However, the loop counter increment (or decrement) must be 1.

```
j IN -5..5
k IN REVERSE first..last
step IN 0..TRUNC(high/low) * 2
```

Internally, PL/SQL assigns the values of the bounds to temporary PLS_INTEGER variables, and, if necessary, rounds the values to the nearest integer. The magnitude range of a PLS_INTEGER is \(-2^{31} .. 2^{31}\). So, if a bound evaluates to a number outside that range, you get a numeric overflow error when PL/SQL attempts the assignment, as the following example shows:

```
DECLARE
    hi NUMBER := 2**32;
BEGIN
    FOR j IN 1..hi LOOP  -- causes a 'numeric overflow' error
        ...
    END LOOP;
END;
```

Some languages provide a STEP clause, which lets you specify a different increment (5 instead of 1 for example). PL/SQL has no such structure, but you can easily build one. Inside the FOR loop, simply multiply each reference to the loop counter by the new increment. In the following example, you assign today’s date to elements 5, 10, and 15 of an index-by table:

```
DECLARE
    TYPE DateList IS TABLE OF DATE INDEX BY BINARY_INTEGER;
dates DateList;
    k CONSTANT INTEGER := 5;  -- set new increment
BEGIN
    FOR j IN 1..3 LOOP
        dates(j*k) := SYSDATE;  -- multiply loop counter by increment
    END LOOP;
    ...
END;
```
Dynamic Ranges
PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
    ...
END LOOP;
```

The value of `emp_count` is unknown at compile time; the `SELECT` statement returns the value at run time.

What happens if the lower bound of a loop range evaluates to a larger integer than the upper bound? As the next example shows, the sequence of statements within the loop is not executed and control passes to the next statement:

```
-- limit becomes 1
FOR i IN 2..limit LOOP
    sequence_of_statements  -- executes zero times
END LOOP;
-- control passes here
```

Scope Rules
The loop counter is defined only within the loop. You cannot reference it outside the loop. After the loop is exited, the loop counter is undefined, as the following example shows:

```
FOR ctr IN 1..10 LOOP
    ...
END LOOP;
sum := ctr - 1;  -- illegal
```

You need not explicitly declare the loop counter because it is implicitly declared as a local variable of type `INTEGER`. The next example shows that the local declaration hides any global declaration:

```
DECLARE
    ctr  INTEGER;
BEGIN
    ...
    FOR ctr IN 1..25 LOOP
        ...
        IF ctr > 10 THEN ...  -- refers to loop counter
            END LOOP;
    END LOOP;
END;
```
Iterative Control: LOOP and EXIT Statements

To reference the global variable in this example, you must use a label and dot notation, as follows:

```
<<main>>
DECLARE
  ctr  INTEGER;
...
BEGIN
  ...
  FOR ctr IN 1..25 LOOP
    ...
    IF main.ctr > 10 THEN  -- refers to global variable
      ...
    END IF;
  END LOOP;
END main;
```

The same scope rules apply to nested FOR loops. Consider the example below. Both loop counters have the same name. So, to reference the outer loop counter from the inner loop, you must use a label and dot notation, as follows:

```
<<outer>>
FOR step IN 1..25 LOOP
  FOR step IN 1..10 LOOP
    ...
    IF outer.step > 15 THEN ...
  END LOOP;
END LOOP outer;
```

**Using the EXIT Statement**

The EXIT statement lets a FOR loop complete prematurely. For example, the following loop normally executes ten times, but as soon as the FETCH statement fails to return a row, the loop completes no matter how many times it has executed:

```
FOR j IN 1..10 LOOP
  FETCH c1 INTO emp_rec;
  EXIT WHEN c1%NOTFOUND;
  ...
END LOOP;
```

Suppose you must exit from a nested FOR loop prematurely. You can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that
you want to complete. Then, use the label in an EXIT statement to specify which
FOR loop to exit, as follows:

```plsql
<<outer>>
FOR i IN 1..5 LOOP
  ...
  FOR j IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT outer WHEN c1%NOTFOUND;  -- exit both FOR loops
    ...
  END LOOP;
END LOOP outer;
-- control passes here
```

### 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 structure of PL/SQL is such that 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 complex, unstructured code (sometimes
called *spaghetti code*) that is hard to understand and maintain. So, 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.

### 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. In the following example, you go to an executable statement farther down in
a sequence of statements:

```plsql
BEGIN
  ...
  GOTO insert_row;
  ...
  <<insert_row>>
    INSERT INTO emp VALUES ...
END;
```
In the next example, you go to a PL/SQL block farther up in a sequence of statements:

```
BEGIN
...
<<update_row>>
BEGIN
  UPDATE emp SET ...
  ...
END;
...
GOTO update_row;
...
END;
```

The label `end_loop` in the following example is illegal because it does not precede an executable statement:

```
DECLARE
  done  BOOLEAN;
BEGIN
...
FOR i IN 1..50 LOOP
  IF done THEN
    GOTO end_loop;
  END IF;
  ...
<<end_loop>>  -- illegal
END LOOP;  -- not an executable statement
END;
```

To debug the last example, just add the `NULL` statement, as follows:

```
FOR i IN 1..50 LOOP
  IF done THEN
    GOTO end_loop;
  END IF;
  ...
<<end_loop>>
NULL;  -- an executable statement
END LOOP;
```

As the following example shows, a `GOTO` statement can branch to an enclosing block from the current block:

```
DECLARE
```

4-18   PL/SQL User's Guide and Reference
my_ename CHAR(10);
BEGIN
  SELECT ename INTO my_ename FROM emp WHERE ...
  BEGIN
    ...
    GOTO get_name;  -- branch to enclosing block
    END;
  END;
END;

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

Restrictions
Some possible destinations of a GOTO statement are illegal. Specifically, a GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement, or sub-block. For example, the following GOTO statement is illegal:

BEGIN
  ...
  GOTO update_row;  -- illegal branch into IF statement
  ...
  IF valid THEN
    ...
    <<update_row>>
    UPDATE emp SET ...
    END IF;
  END;
END;

As the example below shows, a GOTO statement cannot branch from one IF statement clause to another. Likewise, a GOTO statement cannot branch from one CASE statement WHEN clause to another.

BEGIN
  ...
  IF valid THEN
    ...
    GOTO update_row;  -- illegal branch into ELSE clause
  ELSE
    ...
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;
The next example shows that a GOTO statement cannot branch from an enclosing block into a sub-block:

```
BEGIN
  ...
  IF status = 'OBSOLETE' THEN
    GOTO delete_part;  -- illegal branch into sub-block
  END IF;
  ...
BEGIN
  ...
  <delete_part>
    DELETE FROM parts WHERE ...
  END;
END;
```

Also, a GOTO statement cannot branch out of a subprogram, as the following example shows:

```
DECLARE
  ...
PROCEDURE compute_bonus (emp_id NUMBER) IS
  BEGIN
    ...
    GOTO update_row;  -- illegal branch out of subprogram
  END;
BEGIN
  ...
  <update_row>
    UPDATE emp SET ...
  END;
```

Finally, a GOTO statement cannot branch from an exception handler into the current block. For example, the following GOTO statement is illegal:

```
DECLARE
  ...
  pe_ratio  REAL;
BEGIN
  ...
  SELECT price / NVL(earnings, 0) INTO pe_ratio FROM ...
  <insert_row>
  INSERT INTO stats VALUES (pe_ratio, ...);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
```
pe_ratio := 0;
    GOTO insert_row;  -- illegal branch into current block
END;

However, a GOTO statement can branch from an exception handler into an enclosing block.

**NULL Statement**

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. In a construct allowing alternative actions, the NULL statement serves as a placeholder. It tells readers that the associated alternative has not been overlooked, but that indeed no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

```plaintext
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        ROLLBACK;
    WHEN VALUE_ERROR THEN
        INSERT INTO errors VALUES ... COMMIT;
    WHEN OTHERS THEN
        NULL;
END;
```

Each clause in an IF statement must contain at least one executable statement. The NULL statement is executable, so you can use it in clauses that correspond to circumstances in which no action is taken. In the following example, the NULL statement emphasizes that only top-rated employees get bonuses:

```plaintext
IF rating > 90 THEN
    compute_bonus(emp_id);
ELSE
    NULL;
END IF;
```

Also, the NULL statement is a handy way to create stubs when designing applications from the top down. A stub is dummy subprogram that lets you defer the definition of a procedure or function until you test and debug the main program. In the following example, the NULL statement meets the requirement that at least one statement must appear in the executable part of a subprogram:

```plaintext
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
```
BEGIN
    NULL;
END debit_account;
Knowledge is that area of ignorance that we arrange and classify. —Ambrose Bierce

Many programming techniques use collection types such as arrays, bags, lists, nested tables, sets, and trees. To support these techniques in database applications, PL/SQL provides the datatypes TABLE and VARRAY, which allow you to declare index-by tables, nested tables and variable-size arrays. In this chapter, you learn how those types let you reference and manipulate collections of data as whole objects. You also learn how the datatype RECORD lets you treat related but dissimilar data as a logical unit.

This chapter discusses the following topics:

What Is a Collection?
Defining and Declaring Collections
Initializing and Referencing Collections
Assigning and Comparing Collections
Manipulating Collections
Using Collection Methods
Avoiding Collection Exceptions
Reducing Loop Overhead for Collections with Bulk Binds
What Is a Record?
Defining and Declaring Records
Initializing Records
Assigning Records
Manipulating Records
What Is a Collection?

A collection is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers two collection types. Items of type TABLE are either index-by tables (Version 2 PL/SQL tables) or nested tables (which extend the functionality of index-by tables). Items of type VARRAY are varrays (short for variable-size arrays).

Collections work like the arrays found in most third-generation programming languages. Although collections can have only one dimension, you can model multi-dimensional arrays by creating collections whose elements are also collections.

You can define collection types in a package, then use them programmatically in your applications. Also, you can pass collections as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. In addition, collections can store instances of an object type and (except for index-by tables) can be attributes of an object type.

Understanding Nested Tables

Within the database, nested tables can be considered one-column database tables. Oracle stores the rows of a nested table in no particular order. But, when you retrieve the nested table into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. That gives you array-like access to individual rows.

PL/SQL nested tables are like one-dimensional arrays. You can model multi-dimensional arrays by creating nested tables whose elements are also nested tables.

Nested tables differ from arrays in two important ways:

1. Arrays have a fixed upper bound, but nested tables are unbounded (see Figure 5–1). So, the size of a nested table can increase dynamically.
Arrays must be dense (have consecutive subscripts). So, you cannot delete individual elements from an array. Initially, nested tables are dense, but they can become sparse (have nonconsecutive subscripts). So, you can delete elements from a nested table using the built-in procedure DELETE. That might leave gaps in the index, but the built-in function NEXT lets you iterate over any series of subscripts.

**Nested Tables Versus Index-By Tables**

Index-by tables and nested tables are similar. For example, they have the same structure, and their individual elements are accessed in the same way (using subscript notation). The main difference is that nested tables can be stored in a database column (hence the term “nested table”) but index-by tables cannot.

Nested tables extend the functionality of index-by tables by letting you SELECT, INSERT, UPDATE, and DELETE nested tables stored in the database. (Remember, index-by tables cannot be stored in the database.) Also, some collection methods operate only on nested tables and varrays. For example, the built-in procedure TRIM cannot be applied to index-by tables.

Another advantage of nested tables is that an uninitialized nested table is atomically null (that is, the table itself is null, not its elements), but an uninitialized index-by table is merely empty. So, you can apply the IS NULL comparison operator to nested tables but not to index-by tables.

However, index-by tables also have some advantages. For example, PL/SQL supports implicit (automatic) datatype conversion between host arrays and index-by tables (but not nested tables). So, the most efficient way to pass collections to and from the database server is to use anonymous PL/SQL blocks to bulk-bind input and output host arrays to index-by tables.
Also, index-by tables are initially sparse. So, they are convenient for storing reference data using a numeric primary key (account numbers or employee numbers for example) as the index.

In some (relatively minor) ways, index-by tables are more flexible than nested tables. For example, index-by tables can have negative subscripts; nested tables cannot. Also, some element types are allowed for index-by tables but not for nested tables (see "Referencing Collection Elements" on page 5-10). Finally, to extend a nested table, you must use the built-in procedure `EXTEND`, but to extend an index-by table, you just specify larger subscripts.

**Understanding Varrays**

Items of type `VARRAY` are called *varrays*. They allow you to associate a single identifier with an entire collection. This association lets you manipulate the collection as a whole and reference individual elements easily. To reference an element, you use standard subscripting syntax (see Figure 5–2). For example, `Grade(3)` references the third element in varray `Grades`.

![Figure 5–2  Varray of Size 10](image)

A varray has a maximum size, which you must specify in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. For example, the current upper bound for varray `Grades` is 7, but you can extend it to 8, 9, 10, and so on. Thus, a varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

**Varrays Versus Nested Tables**

Nested tables differ from varrays in the following ways:

- Varrays have a maximum size, but nested tables do not.
- Varrays are always dense, but nested tables can be sparse. So, you can delete individual elements from a nested table but not from a varray.
- By default, Oracle stores varray data in-line (in the same table) unless it exceeds 4K, in which case the data is stored out-of-line (but in the same tablespace).
Oracle stores nested table data out-of-line in a *store table*, which is a system-generated database table associated with the nested table.

- When stored in the database, varrays retain their ordering and subscripts, but nested tables do not.

Which collection type should you use? That depends on your wants and the size of the collection. A varray is stored as an opaque object, whereas a nested table is stored in a storage table with every element mapped to a row in the table. So, if you want efficient queries, use nested tables. If you want to retrieve entire collections as a whole, use varrays. However, when collections get very large, it becomes impractical to retrieve all elements at once. So, varrays are better suited for small collections.

**Defining and Declaring Collections**

To create collections, you define a collection type, then declare collections of that type. You can define TABLE and VARRAY types in the declarative part of any PL/SQL block, subprogram, or package. For nested tables, use the syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
```

and for varrays, use the syntax:

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit)
  OF element_type [NOT NULL];
```

where `type_name` is a type specifier used later to declare collections, `size_limit` is a positive integer literal, and `element_type` is any PL/SQL datatype except

- `BINARY_INTEGER`, `PLS_INTEGER`
- `BOOLEAN`
- `BLOB`, `CLOB` (restriction applies only to varrays)
- `LONG`, `LONG RAW`
- `NATURAL`, `NATURALN`
- `NCHAR`, `NCLOB`, `NVARCHAR2`
- object types with `BLOB` or `CLOB` attributes (restriction applies only to varrays)
- object types with `TABLE` or `VARRAY` attributes
- `POSITIVE`, `POSITIVEN`
- `REF CURSOR`
- `SIGNTYPE`
- `STRING`

For index-by tables, use the syntax
Defining and Declaring Collections

**PL/SQL User's Guide and Reference**

**TYPE** type_name IS TABLE OF element_type [NOT NULL]  
INDEX BY BINARY_INTEGER;

Unlike nested tables and varrays, index-by tables can have the following element types: BINARY_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, PLS_INTEGER, POSITIVE, POSITIVEN, SIGTYPE, and STRING.

Index-by tables are initially sparse. That enables you, for example, to store reference data in an index-by table using a numeric primary key as the index. In the example below, you declare an index-by table of records. Each element of the table stores a row from the emp database table.

```
DECLARE  
    TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE  
        INDEX BY BINARY_INTEGER;  
    emp_tab EmpTabTyp;  
BEGIN  
    /* Retrieve employee record. */  
    SELECT * INTO emp_tab(7468) FROM emp WHERE empno = 7468;  
    ...  
END;
```

When defining a **VARRAY** type, you must specify its maximum size. In the following example, you define a type that stores up to 366 dates:

```
DECLARE  
    TYPE Calendar IS VARRAY(366) OF DATE;
```

To specify the element type, you can use **%TYPE**, which provides the datatype of a variable or database column. Also, you can use **%ROWTYPE**, which provides the rowtype of a cursor or database table. Two examples follow:

```
DECLARE  
    TYPE EmpList IS TABLE OF emp.ename%TYPE; -- based on column  
    CURSOR c1 IS SELECT * FROM dept;  
    TYPE DeptFile IS VARRAY(20) OF c1%ROWTYPE; -- based on cursor
```

In the next example, you use a **RECORD** type to specify the element type:

```
DECLARE  
    TYPE AnEntry IS RECORD (  
        term   VARCHAR2(20),  
        meaning   VARCHAR2(200));  
    TYPE Glossary IS VARRAY(250) OF AnEntry;
```

In the final example, you impose a **NOT NULL** constraint on the element type:
DECLARE
    TYPE EmpList IS TABLE OF emp.empno%TYPE NOT NULL;

An initialization clause is not required (or allowed).

Declaring Collections

Once you define a collection type, you can declare collections of that type, as the following SQL*Plus script shows:

CREATE TYPE CourseList AS TABLE OF VARCHAR2(10)  -- define type
/
CREATE TYPE Student AS OBJECT (  -- create object
    id_num  INTEGER(4),
    name    VARCHAR2(25),
    address VARCHAR2(35),
    status  CHAR(2),
    courses CourseList)  -- declare nested table as attribute
/

The identifier courses represents an entire nested table. Each element of courses will store the code name of a college course such as 'Math 1020'.

The script below creates a database column that stores varrays. Each element of the varrays will store a Project object.

CREATE TYPE Project AS OBJECT(  --create object
    project_no NUMBER(2),
    title      VARCHAR2(35),
    cost       NUMBER(7,2))
/
CREATE TYPE ProjectList AS VARRAY(50) OF Project  -- define VARRAY type
/
CREATE TABLE department (  -- create database table
    dept_id  NUMBER(2),
    name     VARCHAR2(15),
    budget   NUMBER(11,2),
    projects ProjectList)  -- declare varray as column
/

The following example shows that you can use %TYPE to provide the datatype of a previously declared collection:

DECLARE
    TYPE Platoon IS VARRAY(20) OF Soldier;
You can declare collections as the formal parameters of functions and procedures. That way, you can pass collections to stored subprograms and from one subprogram to another. In the following example, you declare a nested table as the formal parameter of a packaged procedure:

```sql
CREATE PACKAGE personnel AS
    TYPE Staff IS TABLE OF Employee;
    ...
    PROCEDURE award_bonuses (members IN Staff);
END personnel;
```

Also, you can specify a collection type in the `RETURN` clause of a function specification, as the following example shows:

```sql
DECLARE
    TYPE SalesForce IS VARRAY(25) OF Salesperson;
    FUNCTION top_performers (n INTEGER) RETURN SalesForce IS ...
```

Collections follow the usual scoping and instantiation rules. In a block or subprogram, collections are instantiated when you enter the 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.

### Initializing and Referencing Collections

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" collections from the elements passed to it. In the following example, you pass six elements to constructor `CourseList()`, which returns a nested table containing those elements:

```sql
DECLARE
    my_courses CourseList;
BEGIN
    my_courses := CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100', 'PoSc 3141', 'Mktg 3312', 'Engl 2005');
    ...
END;
```
In the next example, you pass three objects to constructor ProjectList(), which returns a varray containing those objects:

```sql
DECLARE
    accounting_projects ProjectList;
BEGIN
    accounting_projects :=
        ProjectList(Project(1, 'Design New Expense Report', 3250),
                    Project(2, 'Outsource Payroll', 12350),
                    Project(3, 'Audit Accounts Payable', 1425));
...
END;
```

You need not initialize the whole varray. For example, if a varray has a maximum size of 50, you can pass fewer than 50 elements to its constructor.

Unless you impose the NOT NULL constraint or specify a record type for elements, you can pass null elements to a constructor. An example follows:

```sql
BEGIN
    my_courses := CourseList('Math 3010', NULL, 'Stat 3202', ...);
```

The next example shows that you can initialize a collection in its declaration, which is a good programming practice:

```sql
DECLARE
    my_courses CourseList :=
        CourseList('Art 1111', 'Hist 3100', 'Engl 2005', ...);
```

If you call a constructor without arguments, you get an empty but non-null collection, as the following example shows:

```sql
DECLARE
    TYPE Clientele IS VARRAY(100) OF Customer;
    vips Clientele := Clientele(); -- initialize empty varray
BEGIN
    IF vips IS NOT NULL THEN  -- condition yields TRUE
        ...
    END IF;
END;
```

PL/SQL never calls a constructor implicitly, so you must call it explicitly. (That does not apply to index-by tables, which do not have constructors.) Constructor calls are allowed wherever function calls are allowed.
In the example below, you insert a Student object into object table sophomores. The table constructor CourseList() provides a value for attribute courses.

```plsql
BEGIN
    INSERT INTO sophomores
        VALUES (Student(5035, 'Janet Alvarez', '122 Broad St', 'FT',
            CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100', ...)));

```...

In the final example, you insert a row into database table department. The varray constructor ProjectList() provides a value for column projects.

```plsql
BEGIN
    INSERT INTO department
        VALUES(60, 'Security', 750400,
            ProjectList(Project(1, 'Issue New Employee Badges', 9500),
                Project(2, 'Find Missing IC Chips', 2750),
                Project(3, 'Inspect Emergency Exits', 1900)));

```...

### 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 syntax

`collection_name(subscript)`

where `subscript` is an expression that yields an integer. For index-by tables, the legal subscript range is $-2^{31} \ldots 2^{31}$. For nested tables, the legal range is $1 \ldots 2^{31}$. And, for varrays, the legal range is $1 \ldots \text{size_limit}$.

You can reference a collection in all expression contexts. In the following example, you reference an element in nested table names:

```plsql
DECLARE
    TYPE Roster IS TABLE OF VARCHAR2(15);
    names Roster := Roster('J Hamil', 'D Caruso', 'R Singh', ...);
    i BINARY_INTEGER;
BEGIN
    ...
    IF names(i) = 'J Hamil' THEN
        ...
    END IF;
END;

```
The next example shows that you can reference the elements of a collection in subprogram calls:

DECLARE
  TYPE Roster IS TABLE OF VARCHAR2(15);
  names Roster := Roster('J Hamil','D Piro','R Singh',...);
  i BINARY_INTEGER;
BEGIN
  ...  
  verify_name(names(i)); -- call procedure
END;

When calling a function that returns a collection, you can use the following syntax to reference an element in the returned collection, if you do not want to store the entire return value:

function_name(parameter_list)(subscript)

For example, the following call references the third element in the varray returned by function new_hires:

DECLARE
  TYPE Staff IS VARRAY(20) OF Employee;
  staffer Employee;
  FUNCTION new_hires (hiredate DATE) RETURN Staff IS ... 
BEGIN
  staffer := new_hires('16-OCT-96')(3); -- 3rd element of varray
  ...
END;

Assigning and Comparing Collections

One collection can be assigned to another by an INSERT, UPDATE, FETCH, or SELECT statement, an assignment statement, or a subprogram call. As the example below shows, the collections must have the same datatype. Having the same element type is not enough.

DECLARE
  TYPE Clientele IS VARRAY(100) OF Customer;
  TYPE Vips IS VARRAY(100) OF Customer;
  group1 Clientele := Clientele(...);
  group2 Clientele := Clientele(...);
  group3 Vips := Vips(...);
BEGIN
  group2 := group1;

Assigning and Comparing Collections

```
group3 := group2;  -- not allowed; different datatypes
```

If you assign an atomically null collection to another collection, the other collection becomes atomically null (and must be reinitialized). Consider the following example:

```
DECLARE
    TYPE Clientele IS TABLE OF Customer;
    group1 Clientele := Clientele(...);  -- initialized
    group2 Clientele;  -- atomically null
BEGIN
    IF group1 IS NULL THEN ...  -- condition yields FALSE
        group1 := group2;
    IF group1 IS NULL THEN ...  -- condition yields TRUE
        ...
END;
```

Likewise, if you assign the non-value NULL to a collection, the collection becomes atomically null.

Assigning Collection Elements

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. If `subscript` is null or not convertible to an integer, PL/SQL raises the predefined exception `VALUE_ERROR`. 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`. Some examples follow:

```
DECLARE
    TYPE NumList IS TABLE OF INTEGER;
    nums NumList;
BEGIN
    /* Assume execution continues despite the raised exceptions. */
    nums(1) := 10;  -- raises COLLECTION_IS_NULL
    nums := NumList(10,20,30);
    nums(1) := ASCII('A');
    nums(2) := 10 * nums(1);
    nums('B') := 15;  -- raises VALUE_ERROR
    nums(4) := 40;  -- raises SUBSCRIPT_BEYOND_COUNT
END;
```
Comparing Whole Collections

Nested tables and varrays can be atomically null, so they can be tested for nullity, as the following example shows:

```plsql
DECLARE
    TYPE Staff IS TABLE OF Employee;
    members Staff;
BEGIN
    ...
    IF members IS NULL THEN ... -- condition yields TRUE;
END;
```

However, collections cannot be compared for equality or inequality. For instance, the following IF condition is not allowed:

```plsql
DECLARE
    TYPE Clientele IS TABLE OF Customer;
    group1 Clientele := Clientele(...);
    group2 Clientele := Clientele(...);
BEGIN
    ...
    IF group1 = group2 THEN -- causes compilation error
        ...
    END IF;
END;
```

This restriction also applies to implicit comparisons. For example, collections cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` list.

Manipulating Collections

Within PL/SQL, collections add flexibility and procedural power. A big advantage is that your program can compute subscripts to process specific elements. A bigger advantage is that the program can use SQL to manipulate in-memory collections.

Some Nested Table Examples

In SQL*Plus, suppose you define object type `Course`, as follows:

```sql
SQL> CREATE TYPE Course AS OBJECT (
    2    course_no NUMBER(4),
    3    title     VARCHAR2(35),
    4    credits   NUMBER(1));
```
Next, you define TABLE type `CourseList`, which stores `Course` objects:

```sql
SQL> CREATE TYPE CourseList AS TABLE OF Course;
```

Finally, you create database table `department`, which has a column of type `CourseList`, as follows:

```sql
SQL> CREATE TABLE department (  
    name VARCHAR2(20),  
    director VARCHAR2(20),  
    office VARCHAR2(20),  
    courses CourseList  
)  
  NESTED TABLE courses STORE AS courses_tab;
```

Each item in column `courses` is a nested table that will store the courses offered by a given department. The `NESTED TABLE` clause is required because `department` has a nested table column. The clause identifies the nested table and names a system-generated store table, in which Oracle stores data out-of-line.

Now, you can populate database table `department`. In the following example, notice how table constructor `CourseList()` provides values for column `courses`:

```sql
BEGIN  
  INSERT INTO department  
  VALUES('Psychology', 'Irene Friedman', 'Fulton Hall 133',  
         CourseList(Course(1000, 'General Psychology', 5),  
                    Course(2100, 'Experimental Psychology', 4),  
                    Course(2200, 'Psychological Tests', 3),  
                    Course(2250, 'Behavior Modification', 4),  
                    Course(3540, 'Groups and Organizations', 3),  
                    Course(3552, 'Human Factors in Business', 4),  
                    Course(4210, 'Theories of Learning', 4),  
                    Course(4320, 'Cognitive Processes', 4),  
                    Course(4410, 'Abnormal Psychology', 4)));
  INSERT INTO department  
  VALUES('History', 'John Whalen', 'Applegate Hall 142',  
         CourseList(Course(1011, 'History of Europe I', 4),  
                    Course(1012, 'History of Europe II', 4),  
                    Course(1202, 'American History', 5),  
                    Course(2130, 'The Renaissance', 3),  
                    Course(2132, 'The Reformation', 3),  
                    Course(3105, 'History of Ancient Greece', 4),  
                    Course(3321, 'Early Japan', 4),  
                    Course(3601, 'Latin America Since 1825', 4),  
                    Course(3702, 'Medieval Islamic History', 4)));
END;
```
INSERT INTO department
VALUES('English', 'Lynn Saunders', 'Breakstone Hall 205',
CourseList(Course(1002, 'Expository Writing', 3),
Course(2020, 'Film and Literature', 4),
Course(2418, 'Modern Science Fiction', 3),
Course(2810, 'Discursive Writing', 4),
Course(3010, 'Modern English Grammar', 3),
Course(3720, 'Introduction to Shakespeare', 4),
Course(3760, 'Modern Drama', 4),
Course(3822, 'The Short Story', 4),
Course(3870, 'The American Novel', 5)));
END;

In the following example, you revise the list of courses offered by the English Department:

DECLARE
    new_courses CourseList :=
        CourseList(Course(1002, 'Expository Writing', 3),
                    Course(2020, 'Film and Literature', 4),
                    Course(2810, 'Discursive Writing', 4),
                    Course(3010, 'Modern English Grammar', 3),
                    Course(3550, 'Realism and Naturalism', 4),
                    Course(3720, 'Introduction to Shakespeare', 4),
                    Course(3760, 'Modern Drama', 4),
                    Course(3822, 'The Short Story', 4),
                    Course(3870, 'The American Novel', 4),
                    Course(4210, '20th-Century Poetry', 4),
                    Course(4725, 'Advanced Workshop in Poetry', 5));
BEGIN
    UPDATE department
    SET courses = new_courses WHERE name = 'English';
END;

In the next example, you retrieve all the courses offered by the Psychology Department into a local nested table:

DECLARE
    psyc_courses CourseList;
BEGIN
    SELECT courses INTO psyc_courses FROM department
    WHERE name = 'Psychology';
    ...
END;
Some Varray Examples

In SQL*Plus, suppose you define object type Project, as follows:

```sql
SQL> CREATE TYPE Project AS OBJECT (
2  project_no NUMBER(2),
3  title      VARCHAR2(35),
4  cost       NUMBER(7,2));
```

Next, you define VARRAY type ProjectList, which stores Project objects:

```sql
SQL> CREATE TYPE ProjectList AS VARRAY(50) OF Project;
```

Finally, you create relational table department, which has a column of type ProjectList, as follows:

```sql
SQL> CREATE TABLE department (  
2  dept_id  NUMBER(2),
3  name     VARCHAR2(15),
4  budget   NUMBER(11,2),
5  projects ProjectList);
```

Each item in column projects is a varray that will store the projects scheduled for a given department.

Now, you are ready to populate relational table department. In the following example, notice how varray constructor ProjectList() provides values for column projects:

```sql
BEGIN
  INSERT INTO department
    VALUES(30, 'Accounting', 1205700,
           ProjectList(Project(1, 'Design New Expense Report', 3250),
                        Project(2, 'Outsource Payroll', 12350),
                        Project(3, 'Evaluate Merger Proposal', 2750),
                        Project(4, 'Audit Accounts Payable', 1425)));

  INSERT INTO department
    VALUES(50, 'Maintenance', 925300,
           ProjectList(Project(1, 'Repair Leak in Roof', 2850),
                        Project(2, 'Install New Door Locks', 1700),
                        Project(3, 'Wash Front Windows', 975),
                        Project(4, 'Repair Faulty Wiring', 1350),
                        Project(5, 'Winterize Cooling System', 1125)));
END;
```
In the following example, you update the list of projects assigned to the Security Department:

```sql
DECLARE
    new_projects ProjectList :=
        ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                    Project(2, 'Develop New Patrol Plan', 1250),
                    Project(3, 'Inspect Emergency Exits', 1900),
                    Project(4, 'Upgrade Alarm System', 3350),
                    Project(5, 'Analyze Local Crime Stats', 825));
BEGIN
    UPDATE department
    SET projects = new_projects WHERE dept_id = 60;
END;
```

In the next example, you retrieve all the projects for the Accounting Department into a local varray:

```sql
DECLARE
    my_projects ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
    WHERE dept_id = 30;
    ...
END;
```

In the final example, you delete the Accounting Department and its project list from table department:

```sql
BEGIN
    DELETE FROM department WHERE dept_id = 30;
END;
```
Manipulating Individual Collection Elements

So far, you have manipulated whole collections. Within SQL, to manipulate the individual elements of a collection, use the operator TABLE. The operand of TABLE is a subquery that returns a single column value for you to manipulate. That value is a nested table or varray.

In the following example, you add a row to the History Department nested table stored in column courses:

```sql
BEGIN
    INSERT INTO TABLE(SELECT courses FROM department WHERE name = 'History')
    VALUES(3340, 'Modern China', 4);
END;
```

In the next example, you revise the number of credits for two courses offered by the Psychology Department:

```sql
DECLARE
    adjustment INTEGER DEFAULT 1;
BEGIN
    UPDATE TABLE(SELECT courses FROM department
                  WHERE name = 'Psychology')
    SET credits = credits + adjustment
    WHERE course_no IN (2200, 3540);
END;
```

In the following example, you retrieve the number and title of a specific course offered by the History Department:

```sql
DECLARE
    my_course_no NUMBER(4);
    my_title VARCHAR2(35);
BEGIN
    SELECT course_no, title INTO my_course_no, my_title
    FROM TABLE(SELECT courses FROM department
                WHERE name = 'History')
    WHERE course_no = 3105;
    ...
END;
```
In the next example, you delete all 5-credit courses offered by the English Department:

```
BEGIN
    DELETE TABLE(SELECT courses FROM department
                  WHERE name = 'English')
    WHERE credits = 5;
END;
```

In the following example, you retrieve the title and cost of the Maintenance Department’s fourth project from the varray column `projects`:

```
DECLARE
    my_cost  NUMBER(7,2);
    my_title VARCHAR2(35);
BEGIN
    SELECT cost, title INTO my_cost, my_title
    FROM TABLE(SELECT projects FROM department
                WHERE dept_id = 50)
    WHERE project_no = 4;
    ...
END;
```

Currently, you cannot reference the individual elements of a varray in an INSERT, UPDATE, or DELETE statement. So, you must use PL/SQL procedural statements. In the following example, stored procedure `add_project` inserts a new project into a department’s project list at a given position:

```
CREATE PROCEDURE add_project (dept_no IN NUMBER,
                               new_project IN Project,
                               position IN NUMBER) AS
    my_projects ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
    WHERE dept_no = dept_id FOR UPDATE OF projects;
    my_projects.EXTEND;  -- make room for new project
    /* Move varray elements forward. */
    FOR i IN REVERSE position..my_projects.LAST - 1 LOOP
        my_projects(i + 1) := my_projects(i);
    END LOOP;
    my_projects(position) := new_project;  -- add new project
    UPDATE department SET projects = my_projects
    WHERE dept_no = dept_id;
END add_project;
```
The following stored procedure updates a given project:

```
CREATE PROCEDURE update_project (  
department_no IN NUMBER,  
proj_no IN NUMBER,  
new_title IN VARCHAR2 DEFAULT NULL,  
new_cost IN NUMBER DEFAULT NULL) AS  
my_projects ProjectList;  
BEGIN  
SELECT projects INTO my_projects FROM department  
WHERE dept_no = dept_id FOR UPDATE OF projects;  
/* Find project, update it, then exit loop immediately. */  
FOR i IN my_projects.FIRST..my_projects.LAST LOOP  
IF my_projects(i).project_no = proj_no THEN  
IF new_title IS NOT NULL THEN  
my_projects(i).title := new_title;  
END IF;  
IF new_cost IS NOT NULL THEN  
my_projects(i).cost := new_cost;  
END IF;  
EXIT;  
END IF;  
END LOOP;  
UPDATE department SET projects = my_projects  
WHERE dept_no = dept_id;  
END update_project;
```

### Manipulating Local Collections

Within PL/SQL, to manipulate a local collection, use the operators **TABLE** and **CAST**. The operands of **CAST** are a collection declared locally (in a PL/SQL anonymous block for example) and a SQL collection type. **CAST** converts the local collection to the specified type. That way, you can manipulate the collection as if it were a SQL database table. In the following example, you count the number of differences between a revised course list and the original (notice that the number of credits for course 3720 changed from 4 to 3):

```
DECLARE  
revised CourseList :=  
CourseList(Course(1002, 'Expository Writing', 3),  
Course(2020, 'Film and Literature', 4),  
Course(2810, 'Discursive Writing', 4),  
Course(3010, 'Modern English Grammar', 3),  
Course(3550, 'Realism and Naturalism', 4),  
```
Using Multilevel Collections

In addition to collections of scalar or object types, you can also create collections whose elements are collections. For example, you can create a nested table of varrays, a varray of varrays, a varray of nested tables, and so on.

Here are some examples showing the syntax and possibilities for multilevel collections.

Multilevel VARRAY Example

declare
    type t1 is varray(10) of integer;
    type ntl1 is varray(10) of t1; -- multilevel varray type
    va t1 := t1(2,3,5);
    -- initialize multilevel varray
    nva ntl1 := ntl1(va, t1(55,6,73), t1(2,4), va);
    i integer;
    val t1;
begin
    -- multilevel access
    i := nva(2)(3); -- i will get value 73
    dbms_output.put_line(i);
    -- add a new varray element to nva
    nva.extend;
    nva(5) := t1(56, 32);
    -- replace an inner varray element
Using Multilevel Collections

```plsql
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;
/
```

Multilevel Table Example

```plsql
declare
  type tbl is table of varchar2(20);
  type ntbl is table of tbl;  -- table of table elements
  type tvl is varray(10) of integer;
  type ntb2 is table of tvl;  -- table of varray elements

  vtbl tbl := tbl('one', 'three');
  vntbl ntbl := ntbl(vtbl);
  vntb2 ntb2 := ntb2(tvl(3,5), tvl(5,7,3));  -- table of varray elements
begin
  vntbl.extend;
  vntbl(2) := vntbl(1);
  -- delete the first element in vntbl
  vntbl.delete(1);
  -- delete the first string from the second table in the nested table
  vntbl(2).delete(1);
end;
/
```

Multilevel Index-By Table Example

```plsql
declare
  type tbl is table of integer index by binary_integer;
  -- the following is index-by table of index-by tables
  type ntbl is table of tbl index by binary_integer;
  type val is varray(10) of varchar2(20);
  -- the following is index-by table of varray elements
  type ntb2 is table of val index by binary_integer;

  vl val := val('hello', 'world');
```
Using Collection Methods

```sql
v2 ntb1;
v3 ntb2;
v4 tb1;
v5 tb1; -- null table
begin
  v4(1) := 34;
  v4(2) := 46456;
  v4(456) := 343;
  v2(23) := v4;
  v3(34) := va1(33, 456, 656, 343);
  -- note that the following assignment raises
  -- ORA-06531: Reference to uninitialized collection
  -- because the element 35 in v2 does not yet exist.
  -- v2(35)(2) := 78;
  -- assign v2(35) to a null table and try again
  v2(35) := v5;
  v2(35)(2) := 78; -- it works now
end;
/
```

Example of Multilevel Collections and Bulk SQL

```sql
create type t1 is varray(10) of integer;
/
create table tab1 (c1 t1);
insert into tab1 values (t1(2,3,5));
insert into tab1 values (t1(9345, 5634, 432453));

declare
type t2 is table of t1;
v2 t2;
begin
  select c1  BULK COLLECT INTO v2 from tab1;
dbms_output.put_line(v2.count); -- prints 2
end;
/
```

Using Collection Methods

The following collection methods help generalize code, make collections easier to use, and make your applications easier to maintain:
A collection method is a built-in function or procedure that operates on collections and is called using dot notation. The syntax follows:

collection_name.method_name[(parameters)]

Collection methods cannot be called from SQL statements. Also, EXTEND and TRIM cannot be used with index-by tables. EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, and NEXT are functions; EXTEND, TRIM, and DELETE are procedures. EXISTS, PRIOR, NEXT, TRIM, EXTEND, and DELETE take integer parameters.

Only EXISTS can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises COLLECTION_IS_NULL.

Using the EXISTS Collection Method

EXISTS(n) returns TRUE if the n-th element in a collection exists. Otherwise, EXISTS(n) returns FALSE. Mainly, you use EXISTS with DELETE to maintain sparse nested tables. You can also use EXISTS to avoid raising an exception when you reference a nonexistent element. In the following example, PL/SQL executes the assignment statement only if element i exists:

IF courses.EXISTS(i) THEN courses(i) := new_course; END IF;

When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

Using the COUNT Collection Method

COUNT returns the number of elements that a collection currently contains. For instance, if varray projects contains 25 elements, the following IF condition is true:

IF projects.COUNT = 25 THEN ...
COUNT is useful because the current size of a collection is not always known. For example, if you fetch a column of Oracle data into a nested table, how many elements does the table contain? COUNT gives you the answer.

You can use COUNT wherever an integer expression is allowed. In the next example, you use COUNT to specify the upper bound of a loop range:

FOR i IN 1..courses.COUNT LOOP ...

For varrays, COUNT always equals LAST. For nested tables, COUNT normally equals LAST. But, if you delete elements from the middle of a nested table, COUNT becomes smaller than LAST.

When tallying elements, COUNT ignores deleted elements.

Using the LIMIT Collection Method

For nested tables, which have no maximum size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain (which you must specify in its type definition). For instance, if the maximum size of varray projects is 25 elements, the following IF condition is true:

IF projects.LIMIT = 25 THEN ...

You can use LIMIT wherever an integer expression is allowed. In the following example, you use LIMIT to determine if you can add 15 more elements to varray projects:

IF (projects.COUNT + 15) < projects.LIMIT THEN ...

Using the FIRST and LAST Collection Methods

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection. If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same index number, as the following example shows:

IF courses.FIRST = courses.LAST THEN ... -- only one element

The next example shows that you can use FIRST and LAST to specify the lower and upper bounds of a loop range provided each element in that range exists:

FOR i IN courses.FIRST..courses.LAST LOOP ...
In fact, you can use FIRST or LAST wherever an integer expression is allowed. In the following example, you use FIRST to initialize a loop counter:

```plsql
i := courses.FIRST;
WHILE i IS NOT NULL LOOP ...
```

For varrays, FIRST always returns 1 and LAST always equals COUNT. For nested tables, FIRST normally returns 1. But, if you delete elements from the beginning of a nested table, FIRST returns a number larger than 1. Also for nested tables, LAST normally equals COUNT. But, 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.

### Using the PRIOR and NEXT Collection 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. Likewise, if n has no successor, NEXT(n) returns NULL.

PRIOR and NEXT do not wrap from one end of a collection to the other. For example, the following statement assigns NULL to n because the first element in a collection has no predecessor:

```plsql
n := courses.PRIOR(courses.FIRST);  -- assigns NULL to n
```

PRIOR is the inverse of NEXT. For instance, if element i exists, the following statement assigns element i to itself:

```plsql
projects(i) := projects.PRIOR(projects.NEXT(i));
```

You can use PRIOR or NEXT to traverse collections indexed by any series of subscripts. In the following example, you use NEXT to traverse a nested table from which some elements have been deleted:

```plsql
i := courses.FIRST;  -- get subscript of first element
WHILE i IS NOT NULL LOOP
    -- do something with courses(i)
    i := courses.NEXT(i);  -- get subscript of next element
END LOOP;
```

When traversing elements, PRIOR and NEXT ignore deleted elements.
Using the EXTEND Collection Method

To increase the size of a collection, use EXTEND. (You cannot use EXTEND with index-by tables.) 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. For example, the following statement appends 5 copies of element 1 to nested table courses:

```
courses.EXTEND(5,1);
```

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.

EXTEND operates on the internal size of a collection, which includes any deleted elements. So, if EXTEND encounters deleted elements, it includes them in its tally. PL/SQL keeps placeholders for deleted elements so that you can replace them if you wish. Consider the following example:

```plsql
DECLARE
    TYPE CourseList IS TABLE OF VARCHAR2(10);
    courses CourseList;
BEGIN
    courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');
    courses.DELETE(3);  -- delete element 3
    /* PL/SQL keeps a placeholder for element 3. So, the next statement appends element 4, not element 3. */
    courses.EXTEND;  -- append one null element
    /* Now element 4 exists, so the next statement does not raise SUBSCRIPT_BEYOND_COUNT. */
    courses(4) := 'Engl 2005';
END;
```

When it includes deleted elements, the internal size of a nested table differs from the values returned by COUNT and LAST. For instance, 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 (whether leading, in the middle, or trailing) are treated alike.

Using the TRIM Collection 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. For example, this statement removes the last three elements from nested table courses:

```
courses.TRIM(3);
```
If \( n \) is greater than \( \text{COUNT} \), \( \text{TRIM}(n) \) raises \text{SUBSCRIPT_BEYOND_COUNT}.

\( \text{TRIM} \) operates on the internal size of a collection. So, if \( \text{TRIM} \) encounters deleted elements, it includes them in its tally. Consider the following example:

```
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, \text{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 \text{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 \( \text{TRIM} \) and \( \text{DELETE} \). It is better to treat nested tables like fixed-size arrays and use only \( \text{DELETE} \), or to treat them like stacks and use only \( \text{TRIM} \) and \( \text{EXTEND} \).

\( \text{PL/SQL} \) does not keep placeholders for trimmed elements. So, you cannot replace a trimmed element simply by assigning it a new value.

**Using the DELETE Collection Method**

This procedure has three forms. \( \text{DELETE} \) removes all elements from a collection. \( \text{DELETE}(n) \) removes the \( n \)th element from an index-by table or nested table. If \( n \) is null, \( \text{DELETE}(n) \) does nothing. \( \text{DELETE}(m,n) \) removes all elements in the range \( m..n \) from an index-by table or nested table. If \( m \) is larger than \( n \) or if \( m \) or \( n \) is null, \( \text{DELETE}(m,n) \) does nothing. Some examples follow:

```
BEGIN
    ...
    courses.DELETE(2);  -- deletes element 2
    courses.DELETE(7,7);  -- deletes element 7
    courses.DELETE(6,3);  -- does nothing
    courses.DELETE(3,6);  -- deletes elements 3 through 6
END;
```

---

5-28  PL/SQL User’s Guide and Reference
Varrays are dense, so you cannot delete their individual elements.

If an element to be deleted does not exist, `DELETE` simply skips it; no exception is raised. PL/SQL keeps placeholders for deleted elements. So, you can replace a deleted element simply by assigning it a new value.

`DELETE` lets you maintain sparse nested tables. In the following example, you retrieve nested table prospects into a temporary table, prune it, then store it back in the database:

```sql
DECLARE
    my_prospects ProspectList;
    revenue      NUMBER;
BEGIN
    SELECT prospects INTO my_prospects FROM customers WHERE ...
    FOR i IN my_prospects.FIRST..my_prospects.LAST LOOP
        estimate_revenue(my_prospects(i), revenue);     -- call procedure
        IF revenue < 25000 THEN
            my_prospects.DELETE(i);
        END IF;
    END LOOP;
    UPDATE customers SET prospects = my_prospects WHERE ...
END;
```

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

### Applying Methods to Collection Parameters

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. So, you can apply the built-in collection methods (`FIRST`, `LAST`, `COUNT`, and so on) to such parameters. In the following example, a nested table is declared as the formal parameter of a packaged procedure:

```sql
CREATE PACKAGE personnel AS
    TYPE Staff IS TABLE OF Employee;
    ...
    PROCEDURE award_bonuses (members IN Staff);
END personnel;
CREATE PACKAGE BODY personnel AS
    ...
    PROCEDURE award_bonuses (members IN Staff) IS
        BEGIN
            ...
            IF members.COUNT > 10 THEN    -- apply method
Avoiding Collection Exceptions

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception. Consider the following example:

```sql
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    nums NumList;  -- atomically null
BEGIN
    /* Assume execution continues despite the raised exceptions. */
    nums(1) := 1;            -- raises COLLECTION_IS_NULL       (1)
    nums := NumList(1,2);  -- initialize table
    nums(NULL) := 3          -- raises VALUE_ERROR              (2)
    nums(0) := 3;            -- raises SUBSCRIPT_OUTSIDE_LIMIT  (3)
    nums(3) := 3;            -- raises SUBSCRIPT_BEYOND_COUNT   (4)
    nums.DELETE(1);  -- delete element 1
    IF nums(1) = 1 THEN ...  -- raises NO_DATA_FOUND            (5)
END;
END personnel;
```

Note: For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

The following list shows when a given exception is raised:

<table>
<thead>
<tr>
<th>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>
<tr>
<td>NO_DATA_FOUND</td>
<td>a subscript designates an element that was deleted</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 legal range</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>a subscript is null or not convertible to an integer</td>
</tr>
</tbody>
</table>

In the first case, the nested table is atomically null. In the second case, the subscript is null. In the third case, the subscript is outside the legal range. In the fourth case, the subscript exceeds the number of elements in the table. In the fifth case, the subscript designates a deleted element.
In some cases, you can pass "invalid" subscripts to a method without raising an exception. For instance, when you pass a null subscript to procedure DELETE, it does nothing. Also, you can replace deleted elements without raising NO_DATA_FOUND, as the following example shows:

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;    -- legal; does not raise NO_DATA_FOUND
    dbms_output.put_line(nums.COUNT);  -- prints 3
END;

Packaged collection types and local collection types are never compatible. For example, suppose you want to call the following packaged procedure:

CREATE PACKAGE pkg1 AS
    TYPE NumList IS VARRAY(25) OF NUMBER;
    PROCEDURE delete_emps (emp_list NumList);
    ...
END pkg1;

CREATE PACKAGE BODY pkg1 AS
    PROCEDURE delete_emps (emp_list NumList) IS ...
    ...
END pkg1;

When you run the PL/SQL block below, the second procedure call fails with a wrong number or types of arguments error. That is because the packaged and local VARRAY types are incompatible even though their definitions are identical.

DECLARE
    TYPE NumList IS VARRAY(25) OF NUMBER(4);
    emps  pkg1.NumList := pkg1.NumList(7369, 7499);
    emps2 NumList := NumList(7521, 7566);
BEGIN
    pkg1.delete_emps(emps);
    pkg1.delete_emps(emps2);  -- causes a compilation error
END;
Reducing Loop Overhead for Collections with Bulk Binds

Embedded in the Oracle RDBMS, the PL/SQL engine accepts any valid PL/SQL block or subprogram. As Figure 5–3 shows, the PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine, which executes the SQL statements and, in some cases, returns data to the PL/SQL engine.

![Diagram of Context Switching](image)

Each context switch between the PL/SQL and SQL engines adds to overhead. So, if many switches are required, performance suffers. That can happen when SQL statements execute inside a loop using collection (index-by table, nested table, varray, or host array) elements as bind variables. For example, the following DELETE statement is sent to the SQL engine with each iteration of the FOR loop:

```plsql
DECLARE
    TYPE NumList IS VARRAY(20) OF NUMBER;
    depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
    ...
    FOR i IN depts.FIRST..depts.LAST LOOP
        DELETE FROM emp WHERE deptno = depts(i);
    END LOOP;
END;
```

In such cases, if the SQL statement affects four or more database rows, the use of bulk binds can improve performance considerably.
How Do Bulk Binds Improve Performance?

The assigning of values to PL/SQL variables in SQL statements is called binding. The binding of an entire collection at once is called bulk binding. Bulk binds improve performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth. For example, the following DELETE statement is sent to the SQL engine just once, with an entire nested table:

```plsql
DECLARE
    TYPE NumList IS VARRAY(20) OF NUMBER;
    depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
    ...  
    FORALL i IN depts.FIRST..depts.LAST
        DELETE FROM emp WHERE deptno = depts(i);
END;
```

In the example below, 5000 part numbers and names are loaded into index-by tables. Then, all table elements are inserted into a database table twice. First, they are inserted using a FOR loop, which completes in 32 seconds. Then, they are bulk-inserted using a FORALL statement, which completes in only 3 seconds.

```sql
SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE parts (pnum NUMBER(4), pname CHAR(15));
Table created.

SQL> GET test.sql
1  DECLARE
2     TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY_INTEGER;
3     TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
4     pnums  NumTab;
5     pnames NameTab;
6     t1 NUMBER(5);
7     t2 NUMBER(5);
8     t3 NUMBER(5);
9     PROCEDURE get_time (t OUT NUMBER) IS
10     BEGIN SELECT TO_CHAR(SYSDATE,'SSSSS') INTO t FROM dual; END;
11  BEGIN
12     FOR j IN 1..5000 LOOP  -- load index-by tables
13         pnums(j) := j;
14         pnames(j) := 'Part No. ' || TO_CHAR(j);
15     END LOOP;
16     get_time(t1);
```

```sql
SQL> GET test.sql
1  DECLARE
2     TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY_INTEGER;
3     TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
4     pnums  NumTab;
5     pnames NameTab;
6     t1 NUMBER(5);
7     t2 NUMBER(5);
8     t3 NUMBER(5);
9     PROCEDURE get_time (t OUT NUMBER) IS
10     BEGIN SELECT TO_CHAR(SYSDATE,'SSSSS') INTO t FROM dual; END;
11  BEGIN
12     FOR j IN 1..5000 LOOP  -- load index-by tables
13         pnums(j) := j;
14         pnames(j) := 'Part No. ' || TO_CHAR(j);
15     END LOOP;
16     get_time(t1);
```
The keyword `FORALL` instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the `FORALL` statement contains an iteration scheme, it is not a `FOR` loop. Its syntax follows:

```
FORALL index IN lower_bound..upper_bound
  sql_statement;
```

The index can be referenced only within the `FORALL` statement and only as a collection subscript. The SQL statement must be an `INSERT`, `UPDATE`, or `DELETE` statement that references collection elements. And, the bounds must specify a valid range of consecutive index numbers. The SQL engine executes the SQL statement once for each index number in the range. As the following example shows, you can use the bounds to bulk-bind arbitrary slices of a collection:

```
DECLARE
  TYPE NumList IS VARRAY(15) OF NUMBER;
  depts NumList := NumList();
BEGIN
  -- fill varray here
```
... FORALL j IN 6..10 -- bulk-bind middle third of varray
    UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
END;

The SQL statement can reference more than one collection. However, the PL/SQL engine bulk-binds only subscripted collections. So, in the following example, it does not bulk-bind the collection `sals`, which is passed to the function `median`:

FORALL i IN 1..20
    INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);

In addition to relational tables, the `FORALL` statement can manipulate object tables, as the following example shows:

CREATE TYPE PNum AS OBJECT (n NUMBER);
CREATE TABLE partno OF PNum;

DECLARE
    TYPE NumTab IS TABLE OF NUMBER;
    nums NumTab := NumTab(1, 2, 3, 4);
    TYPE PNumTab IS TABLE OF PNum;
    pnums PNumTab := PNumTab(PNum(1), PNum(2), PNum(3), PNum(4));
BEGIN
    FORALL i IN pnums.FIRST..pnums.LAST
        INSERT INTO partno VALUES(pnums(i));
    FORALL i IN nums.FIRST..nums.LAST
        DELETE FROM partno WHERE n = 2 * nums(i);
    FORALL i IN nums.FIRST..nums.LAST
        INSERT INTO partno VALUES(100 + nums(i));
END;

**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 follows:

CREATE TABLE emp2 (deptno NUMBER(2), job VARCHAR2(15));

Next, you insert some rows into the table, as follows:
Using the FORALL Statement

```
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(20, 'Bookkeeper');  -- 10-char job title
INSERT INTO emp2 VALUES(30, 'Analyst');
INSERT INTO emp2 VALUES(30, 'Analyst');

Then, you try to append the 7-character string ' (temp)' to certain job titles using the following UPDATE statement:

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20, 30);
BEGIN
    FORALL j IN depts.FIRST..depts.LAST
        UPDATE emp2 SET job = job || ' (temp)'  
        WHERE deptno = depts(j);
    -- raises a "value too large" exception
EXCEPTION
    WHEN OTHERS THEN
        COMMIT;
END;
```

The SQL engine executes the UPDATE statement three times, once for each index number in the specified range, that is, once for depts(10), once for depts(20), and once for depts(30). The first execution succeeds, but the second execution fails because the string value 'Bookkeeper (temp)' is too large for the job column. In this case, only the second execution is rolled back.

When any execution of the SQL statement raises an exception, the FORALL statement halts. In our example, the second execution of the UPDATE statement raises an exception, so the third execution is never done.

Counting Rows Affected by FORALL Iterations with the %BULK_ROWCOUNT Attribute

To process SQL data manipulation statements, the SQL engine opens an implicit cursor named SQL. This cursor's scalar attributes, %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT, return useful information about the most recently executed SQL data manipulation statement.

The SQL cursor has one composite attribute, %BULK_ROWCOUNT, designed for use with the FORALL statement. This attribute has the semantics of an index-by table. Its $i$th element stores the number of rows processed by the $i$th execution of an INSERT,
UPDATE or DELETE statement. If the \(i\)th execution affects no rows, \%BULK_ROWCOUNT\(i\) returns zero. An example follows:

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(10, 20, 50);
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
  -- Did the 3rd UPDATE statement affect any rows?
  IF SQL%BULK_ROWCOUNT(3) = 0 THEN ... END;
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.

\%BULK_ROWCOUNT is usually equal to 1 for inserts, because a typical insert operation affects only a single row. But for the INSERT ... SELECT construct, \%BULK_ROWCOUNT might be greater than 1. For example, the FORALL statement below inserts an arbitrary number of rows for each iteration. After each iteration, \%BULK_ROWCOUNT returns the number of items inserted:

SET SERVEROUTPUT ON;
DECLARE
  TYPE num_tab IS TABLE OF NUMBER;
  deptnums num_tab;
BEGIN
  SELECT deptno BULK COLLECT INTO deptnums FROM DEPT;
  FORALL i IN 1..deptnums.COUNT
    INSERT INTO emp_by_dept
      SELECT empno, deptno FROM emp WHERE deptno = deptnums(i);
  FOR i IN 1..deptnums.COUNT LOOP
    -- Count how many rows were inserted for each department; that is, how many employees are in each department.
    dbms_output.put_line('Dept ' || deptnums(i) || ': inserted ' || SQL%BULK_ROWCOUNT(i) || ' records');
  END LOOP;
  END LOOP;
  dbms_output.put_line('Total records inserted = ' || SQL%ROWCOUNT);
END;
/*
You can also 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.

%FOUND and %NOTFOUND refer only to the last execution of the SQL statement. However, you can use %BULK_ROWCOUNT to infer their values for individual executions. For example, when %BULK_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

Handling FORALL Exceptions with the %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. The syntax follows:

```sql
FORALL index IN lower_bound..upper_bound SAVE EXCEPTIONS
{insert_stmt | update_stmt | delete_stmt}
```

All exceptions raised during the execution are saved in the new cursor attribute %BULK_EXCEPTIONS, which stores a collection of records. Each record has two fields. The first field, %BULK_EXCEPTIONS(i).ERROR_INDEX, holds the "iteration" of the FORALL statement during which the exception was raised. The second field, %BULK_EXCEPTIONS(i).ERROR_CODE, holds the corresponding Oracle error code.

The values stored by %BULK_EXCEPTIONS always refer to the most recently executed FORALL statement. The number of exceptions is saved in the count attribute of %BULK_EXCEPTIONS, that is, %BULK_EXCEPTIONS.COUNT. Its subscripts range from 1 to COUNT.

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.

The following example shows how useful the cursor attribute %BULK_EXCEPTIONS can be:

```sql
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  num_tab NumList := NumList(10,0,11,12,30,0,20,199,2,0,9,1);
  errors NUMBER;
```
BEGIN
...
FORALL i IN num_tab.FIRST..num_tab.LAST SAVE EXCEPTIONS
     DELETE * FROM emp WHERE sal > 500000/num_tab(i);
errors := SQL%BULK_EXCEPTIONS.COUNT;
dbms_output.put_line('Number of errors is ' || 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('Oracle error is ' || SQLERRM(SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
END LOOP;
END;

In this example, PL/SQL raised the predefined exception ZERO_DIVIDE when i equalled 2, 6, 10. After the bulk-bind completed, SQL%BULK_EXCEPTIONS.COUNT returned 3, and the contents of SQL%BULK_EXCEPTIONS were (2,1476), (6,1476), and (10,1476). To get the Oracle error message (which includes the code), we passed SQL%BULK_EXCEPTIONS(i).ERROR_CODE to the error-reporting function SQLERRM. Here is the output:

Number of errors is 3
Error 1 occurred during iteration 2
Oracle error is ORA-01476: divisor is equal to zero
Error 2 occurred during iteration 6
Oracle error is ORA-01476: divisor is equal to zero
Error 3 occurred during iteration 10
Oracle error is ORA-01476: divisor is equal to zero

Restrictions on FORALL

The following restrictions apply to the FORALL statement:

- You can use the FORALL statement only in server-side programs (not in client-side programs). Otherwise, you get the error this feature is not supported in client-side programs.

- The INSERT, UPDATE, or DELETE statement must reference at least one collection. Consider the following example:

```sql
CREATE TABLE pairs (n NUMBER, m NUMBER);

DECLARE
    TYPE NumTab IS TABLE OF NUMBER;
    nums NumTab := NumTab(1, 2, 3);
```
BEGIN
    FORALL j IN nums.FIRST..nums.LAST
        INSERT INTO pairs VALUES(nums(i), 10);  -- works
    FORALL j IN 1..3
        INSERT INTO pairs VALUES(5, 10); -- raises an exception
END;

- All collection elements in the specified range must exist. If an element is missing or was deleted, you get an error. An example follows:

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20, 30, 40);
BEGIN
    depts.DELETE(3);  -- delete third element
    FORALL i IN depts.FIRST..depts.LAST
        DELETE FROM emp WHERE deptno = depts(i);  -- causes an error
END;

- As the following example shows, input collections of composite values cannot be decomposed and bound to database columns:

CREATE TABLE coords (x NUMBER, y NUMBER);
CREATE TYPE Pair AS OBJECT (m NUMBER, n NUMBER);

DECLARE
    TYPE PairTab IS TABLE OF Pair;
    pairs PairTab := PairTab(Pair(1,2), Pair(3,4), Pair(5,6));
    TYPE NumTab IS TABLE OF NUMBER;
    nums NumTab := NumTab(1, 2, 3);
BEGIN
    /* The following statement fails. */
    FORALL i IN 1..3
        UPDATE coords SET (x, y) = pairs(i)
            WHERE x = nums(i);
END;

The workaround is to decompose the composite values manually:

DECLARE
    TYPE PairTab IS TABLE OF Pair;
    pairs PairTab := PairTab(Pair(1,2), Pair(3,4), Pair(5,6));
    TYPE NumTab IS TABLE OF NUMBER;
    nums NumTab := NumTab(1, 2, 3);
BEGIN
    /* The following statement succeeds. */
FORALL i in 1..3
    UPDATE coords SET (x, y) = (pairs(i).m, pairs(i).n)
    WHERE x = nums(i);
END;

Collection subscripts cannot be expressions, as the following example shows:
FORALL j IN mgrs.FIRST..mgrs.LAST
    DELETE FROM emp WHERE mgr = mgrs(j+1);  -- illegal subscript

The cursor attribute \%BULK\_ROWCOUNT cannot be assigned to other collections. Also, it cannot be passed as a parameter to subprograms.

**Retrieving Query Results into Collections with the BULK COLLECT Clause**

The keywords **BULK COLLECT** tell the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. You can use these keywords in the **SELECT INTO**, **FETCH INTO**, and **RETURNING INTO** clauses. Here is the syntax:

```
... BULK COLLECT INTO collection_name[, collection_name] ...
```

The SQL engine bulk-binds all collections referenced in the **INTO** list. The corresponding columns can store scalar or composite values including objects. In the following example, the SQL engine loads the entire empno and ename database columns into nested tables before returning the tables to the PL/SQL engine:

```
DECLARE
    TYPE NumTab IS TABLE OF emp.empno%TYPE;
    TYPE NameTab IS TABLE OF emp.ename%TYPE;
    enums NumTab;  -- no need to initialize
    names NameTab;
BEGIN
    SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;
    ...
END;
```

In the next example, the SQL engine loads all the values in an object column into a nested table before returning the table to the PL/SQL engine:

```
CREATE TYPE Coords AS OBJECT (x NUMBER, y NUMBER);
CREATE TABLE grid (num NUMBER, loc Coords);
INSERT INTO grid VALUES(10, Coords(1,2));
INSERT INTO grid VALUES(20, Coords(3,4));
```
DECLARE
    TYPE CoordsTab IS TABLE OF Coords;
    pairs CoordsTab;
BEGIN
    SELECT loc BULK COLLECT INTO pairs FROM grid;
    -- now pairs contains (1,2) and (3,4)
END;

The SQL engine initializes and extends collections for you. (However, it cannot extend varrays beyond their maximum size.) Then, starting at index 1, it inserts elements consecutively and overwrites any pre-existent elements.

The SQL engine bulk-binds entire database columns. So, if a table has 50,000 rows, the engine loads 50,000 column values into the target collection. However, you can use the pseudocolumn ROWNUM to limit the number of rows processed. In the following example, you limit the number of rows to 100:

DECLARE
    TYPE SalList IS TABLE OF emp.sal%TYPE;
    sals SalList;
BEGIN
    SELECT sal BULK COLLECT INTO sals FROM emp
    WHERE ROWNUM <= 100;
    ...
END;

Example of a Bulk Fetch from a Cursor

The following example shows that you can bulk-fetch from a cursor into one or more collections:

DECLARE
    TYPE NameList IS TABLE OF emp.ename%TYPE;
    TYPE SalList IS TABLE OF emp.sal%TYPE;
    CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
    names NameList;
    sals SalList;
BEGIN
    OPEN c1;
    FETCH c1 BULK COLLECT INTO names, sals;
    ...
END;
Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause

The optional LIMIT clause, allowed only in bulk (not scalar) FETCH statements, lets you limit the number of rows fetched from the database. The syntax is

```sql
FETCH ... BULK COLLECT INTO ... [LIMIT rows];
```

where `rows` can be a literal, variable, or expression but must evaluate to a number. Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`. If the number is not positive, PL/SQL raises `INVALID_NUMBER`. If necessary, PL/SQL rounds the number to the nearest integer.

In the example below, with each iteration of the loop, the FETCH statement fetches ten rows (or less) into index-by table `empnos`. The previous values are overwritten.

```sql
DECLARE
    TYPE NumTab IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    CURSOR c1 IS SELECT empno FROM emp;
    empnos NumTab;
    rows   NATURAL := 10;
BEGIN
    OPEN c1;
    LOOP
        /* The following statement fetches 10 rows (or less). */
        FETCH c1 BULK COLLECT INTO empnos LIMIT rows;
        EXIT WHEN c1%NOTFOUND;
    ... END LOOP;
    CLOSE c1;
END;
```

Retrieving DML Results into a Collection with the RETURNING INTO Clause

You can use the BULK COLLECT clause in the RETURNING INTO clause of an INSERT, UPDATE, or DELETE statement, as the following example shows:

```sql
DECLARE
    TYPE NumList IS TABLE OF emp.empno%TYPE;
    enums NumList;
BEGIN
    DELETE FROM emp WHERE deptno = 20
    RETURNING empno BULK COLLECT INTO enums;
    -- if there were five employees in department 20,
    -- then enums contains five employee numbers
END;
```
Restrictions on BULK COLLECT

The following restrictions apply to the BULK COLLECT clause:

■ You can use the BULK COLLECT clause only in server-side programs (not in client-side programs). Otherwise, you get the error this feature is not supported in client-side programs.

■ All targets in a BULK COLLECT INTO clause must be collections, as the following example shows:

```sql
DECLARE
    TYPE NameList IS TABLE OF emp.ename%TYPE;
    names NameList;
    salary emp.sal%TYPE;
BEGIN
    SELECT ename, sal BULK COLLECT INTO names, salary -- illegal target
        FROM emp WHERE ROWNUM < 50;
    ...
END;
```

■ You cannot bulk-fetch from a cursor into a collection of records, as the following example shows:

```sql
DECLARE
    TYPE DeptRecTab IS TABLE OF dept%ROWTYPE;
    dept_recs DeptRecTab;
    CURSOR c1 IS
        SELECT deptno, dname, loc FROM dept WHERE deptno > 10;
BEGIN
    OPEN c1;
    FETCH c1 BULK COLLECT INTO dept_recs; -- illegal
    ...
END;
```

■ Composite targets (such as objects) cannot be used in the RETURNING INTO clause. Otherwise, you get the error unsupported feature with RETURNING clause.

■ When implicit datatype conversions are needed, multiple composite targets cannot be used in the BULK COLLECT INTO clause.

■ When an implicit datatype conversion is needed, a collection of a composite target (such as a collection of objects) cannot be used in the BULK COLLECT INTO clause.
**Using FORALL and BULK COLLECT Together**

You can combine the `BULK COLLECT` clause with a `FORALL` statement, in which case, the SQL engine bulk-binds column values incrementally. In the following example, if collection `depts` has 3 elements, each of which causes 5 rows to be deleted, then collection `enums` has 15 elements when the statement completes:

```sql
FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp WHERE empno = depts(j)
    RETURNING empno BULK COLLECT INTO enums;
```

The column values returned by each execution are added to the values returned previously. (With a `FOR` loop, the previous values are overwritten.)

You cannot use the `SELECT ... BULK COLLECT` statement in a `FORALL` statement. Otherwise, you get the error `implementation restriction: cannot use FORALL and BULK COLLECT INTO together in SELECT statements`.

**Using Host Arrays with Bulk Binds**

Client-side programs can use anonymous PL/SQL blocks to bulk-bind input and output host arrays. In fact, that 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 Pro*C program and must be prefixed with a colon to distinguish them from PL/SQL collections. In the example below, 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 the host array
    -- and host variables in the host environment
    FORALL i IN :lower..:upper
        DELETE FROM emp WHERE deptno = :depts(i);
    ...
END;
```
What Is a Record?

A record is a group of related data items stored in fields, each with its own name and datatype. Suppose you have various data about an employee such as name, salary, and hire date. These items are logically related but dissimilar in type. A record containing a field for each item lets you treat the data as a logical unit. Thus, records make it easier to organize and represent information.

The attribute %ROWTYPE lets you declare a record that represents a row in a database table. However, you cannot specify the datatypes of fields in the record or declare fields of your own. The datatype RECORD lifts those restrictions and lets you define your own records.

Defining and Declaring Records

To create records, you define a RECORD type, then declare records of that type. You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package using the syntax

```
TYPE type_name IS RECORD (field_declaration[,field_declaration]...);
```

where field_declaration stands for

```
field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
```

and where type_name is a type specifier used later to declare records,
field_type is any PL/SQL datatype except REF CURSOR, and expression yields a value of the same type as field_type.

Note: Unlike VARRAY and (nested) TABLE types, RECORD types cannot be CREATED and stored in the database.

You can use %TYPE and %ROWTYPE to specify field types. In the following example, you define a RECORD type named DeptRec:

```
DECLARE
  TYPE DeptRec IS RECORD (  
    dept_id  dept.deptno%TYPE,
    dept_name VARCHAR2(14),
    dept_loc  VARCHAR2(13));
BEGIN
  ...
END;
```
Notice that field declarations are like variable declarations. Each field has a unique name and specific datatype. So, the value of a record is actually a collection of values, each of some simpler type.

As the example below shows, PL/SQL lets you define records that contain objects, collections, and other records (called nested records). However, object types cannot have attributes of type RECORD.

```plsql
DECLARE
    TYPE TimeRec IS RECORD (    
        seconds SMALLINT,
        minutes SMALLINT,
        hours SMALLINT);
    TYPE FlightRec IS RECORD (    
        flight_no INTEGER,
        plane_id VARCHAR2(10),
        captain Employee, -- declare object
        passengers PassengerList, -- declare varray
        depart_time TimeRec, -- declare nested record
        airport_code VARCHAR2(10));
BEGIN
    ...
END;
```

The next example shows that 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.

```plsql
DECLARE
    TYPE EmpRec IS RECORD (        
        emp_id NUMBER(4),
        last_name VARCHAR2(10),
        dept_num NUMBER(2),
        job_title VARCHAR2(9),
        salary NUMBER(7,2));
    ...
FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRec IS ...
BEGIN
    ...
END;
```
Declaring Records

Once you define a RECORD type, you can declare records of that type, as the example below shows. The identifier item_info represents an entire record.

DECLARE
    TYPE StockItem IS RECORD (
        item_no INTEGER(3),
        description VARCHAR2(50),
        quantity INTEGER,
        price REAL(7,2));
    item_info StockItem;  -- declare record
BEGIN
    ...
END;

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions. An example follows:

DECLARE
    TYPE EmpRec IS RECORD (
        emp_id emp.empno%TYPE,
        last_name VARCHAR2(10),
        job_title VARCHAR2(9),
        salary NUMBER(7,2));
    ...

    PROCEDURE raise_salary (emp_info EmpRec);
BEGIN
    ...
END;

Initializing Records

The example below shows that you can initialize a record in its type definition. When you declare a record of type TimeRec, its three fields assume an initial value of zero.

DECLARE
    TYPE TimeRec IS RECORD (
        secs SMALLINT := 0,
        mins SMALLINT := 0,
        hrs SMALLINT := 0);
BEGIN
    ...
END;
The next example shows that you can impose the **NOT NULL** constraint on any field, and so prevent the assigning of nulls to that field. Fields declared as **NOT NULL** must be initialized.

DECLARE

    TYPE StockItem IS RECORD (
        item_no     INTEGER(3) NOT NULL := 999,
        description VARCHAR2(50),
        quantity    INTEGER,
        price       REAL(7,2));
BEGIN
    ...
END;

### Referencing Records

Unlike elements in a collection, which are accessed using subscripts, fields in a record are accessed by name. To reference an individual field, use dot notation and the following syntax:

`record_name.field_name`

For example, you reference field `hire_date` in record `emp_info` as follows:

`emp_info.hire_date ...`

When calling a function that returns a user-defined record, use the following syntax to reference fields in the record:

`function_name(parameter_list).field_name`

For example, the following call to function `nth_highest_sal` references the field `salary` in record `emp_info`:

DECLARE

    TYPE EmpRec IS RECORD (
        emp_id    NUMBER(4),
        job_title VARCHAR2(9),
        salary    NUMBER(7,2)),
        middle_sal NUMBER(7,2));
    FUNCTION nth_highest_sal (n INTEGER) RETURN EmpRec IS
        emp_info EmpRec;
    BEGIN
        ...
        RETURN emp_info; -- return record
    END;
BEGIN
    middle_sal := nth_highest_sal(10).salary;  -- call function
    ...
END;

When calling a parameterless function, use the following syntax:
function_name().field_name  -- note empty parameter list

To reference nested fields in a record returned by a function, use extended dot
notation. The syntax follows:
function_name(parameter_list).field_name.nested_field_name

For instance, the following call to function item references the nested field
minutes in record item_info:

DECLARE
    TYPE TimeRec IS RECORD (minutes SMALLINT, hours SMALLINT);
    TYPE AgendaItem IS RECORD (
        priority INTEGER,
        subject VARCHAR2(100),
        duration TimeRec);
    FUNCTION item (n INTEGER) RETURN AgendaItem IS
        item_info AgendaItem;
    BEGIN
        ...
        RETURN item_info;  -- return record
    END;
BEGIN
    ...
    IF item(3).duration.minutes > 30 THEN ...  -- call function
END;

Also, use extended dot notation to reference the attributes of an object stored in a
field, as the following example shows:

DECLARE
    TYPE FlightRec IS RECORD ( flight_no INTEGER, plane_id VARCHAR2(10),
        captain Employee, -- declare object
        passengers PassengerList, -- declare varray
        depart_time TimeRec, -- declare nested record
        airport_code VARCHAR2(10));
    flight FlightRec;
BEGIN
  ...
  IF flight.captain.name = 'H Rawlins' THEN ...
END;

**Nulling Records**

To null all the fields in a record, simply assign to it an uninitialized record of the same type, as shown in the following example:

```oracle
DECLARE
  TYPE EmpRec IS RECORD (
    emp_id emp.empno%TYPE,
    job_title VARCHAR2(9),
    salary NUMBER(7,2));
  emp_info EmpRec;
  emp_null EmpRec;
BEGIN
  emp_info.emp_id := 7788;
  emp_info.job_title := 'ANALYST';
  emp_info.salary := 3500;
  emp_info := emp_null;  -- nulls all fields in emp_info
  ...
END;
```

**Assigning Records**

You can assign the value of an expression to a specific field in a record using the following syntax:

```oracle
record_name.field_name := expression;
```

In the following example, you convert an employee name to upper case:

```oracle
emp_info.ename := UPPER(emp_info.ename);
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once. This can be done in two ways. First, you can assign one user-defined record to another if they have the same datatype. Having fields that match exactly is not enough. Consider the following example:

```oracle
DECLARE
  TYPE DeptRec IS RECORD (
    dept_num NUMBER(2),
    dept_name VARCHAR2(14));
  ...
END;
```
As the next example shows, you can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes:

```
DECLARE
    TYPE DeptRec IS RECORD (
        dept_num  NUMBER(2),
        dept_name VARCHAR2(14),
        location  VARCHAR2(13));
    dept1_info DeptRec;
    dept2_info dept%ROWTYPE;
BEGIN
    SELECT * INTO dept2_info FROM dept WHERE deptno = 10;
    dept1_info := dept2_info;
    ...
END;
```

Second, you can use the SELECT or FETCH statement to fetch column values into a record, as the example below shows. The columns in the select-list must appear in the same order as the fields in your record.

```
DECLARE
    TYPE DeptRec IS RECORD (
        dept_num  NUMBER(2),
        dept_name VARCHAR2(14),
        location  VARCHAR2(13));
    dept_info DeptRec;
BEGIN
    SELECT * INTO dept_info FROM dept WHERE deptno = 20;
    ...
END;
```

However, you cannot assign a list of values to a record using an assignment statement. Therefore, the following syntax is illegal:

```
record_name := (value1, value2, value3, ...);  -- illegal
```
The example below shows that you can assign one nested record to another if they have the same datatype. Such assignments are allowed even if the enclosing records have different datatypes.

```
DECLARE
    TYPE TimeRec IS RECORD (mins SMALLINT, hrs SMALLINT);
    TYPE MeetingRec IS RECORD (
        day     DATE,
        time_of TimeRec,  -- nested record
        room_no INTEGER(4));
    TYPE PartyRec IS RECORD (
        day     DATE,
        time_of TimeRec,  -- nested record
        place   VARCHAR2(25));
    seminar MeetingRec;
    party   PartyRec;
BEGIN
    ...
    party.time_of := seminar.time_of;
END;
```

### Comparing Records

Records cannot be tested for nullity, equality, or inequality. For instance, the following IF conditions are not allowed:

```
BEGIN
    ...
    IF emp_info IS NULL THEN ... -- illegal
    IF dept2_info > dept1_info THEN ... -- illegal
END;
```

### Manipulating Records

The datatype RECORD lets you collect information about the attributes of something. The information is easy to manipulate because you can refer to the collection as a whole. In the following example, you collect accounting figures from database tables assets and liabilities, then use ratio analysis to compare the performance of two subsidiary companies:

```
DECLARE
    TYPE FiguresRec IS RECORD (cash REAL, notes REAL, ...);
    sub1_figs FiguresRec;
```
Manipulating Records

```
sub2_figs FiguresRec;
FUNCTION acid_test (figs FiguresRec) RETURN REAL IS ...
BEGIN
  SELECT cash, notes, ... INTO sub1_figs FROM assets, liabilities
  WHERE assets.sub = 1 AND liabilities.sub = 1;
  SELECT cash, notes, ... INTO sub2_figs FROM assets, liabilities
  WHERE assets.sub = 2 AND liabilities.sub = 2;
  IF acid_test(sub1_figs) > acid_test(sub2_figs) THEN ...
  ...
END;
```

Notice how easy it is to pass the collected figures to the function `acid_test`, which computes a financial ratio.

In SQL*Plus, suppose you define object type `Passenger`, as follows:

```
SQL> CREATE TYPE Passenger AS OBJECT(
  2  flight_no NUMBER(3),
  3  name      VARCHAR2(20),
  4  seat      CHAR(5));
```

Next, you define VARRAY type `PassengerList`, which stores `Passenger` objects:

```
SQL> CREATE TYPE PassengerList AS VARRAY(300) OF Passenger;
```

Finally, you create relational table `flights`, which has a column of type `PassengerList`, as follows:

```
SQL> CREATE TABLE flights ( ...
  2  flight_no  NUMBER(3),
  3  gate       CHAR(5),
  4  departure  CHAR(15),
  5  arrival    CHAR(15),
  6  passengers PassengerList);
```

Each item in column `passengers` is a varray that will store the passenger list for a given flight. Now, you can populate database table `flights`, as follows:

```
BEGIN
  INSERT INTO flights
  VALUES(109, '80', 'DFW 6:35PM', 'HOU 7:40PM',
  PassengerList(Passenger(109, 'Paula Trusdale', '13C'),
  Passenger(109, 'Louis Jemenez', '22F'),
  Passenger(109, 'Joseph Braun', '11B'), ...));
  INSERT INTO flights
  VALUES(114, '12B', 'SFO 9:45AM', 'LAX 12:10PM',
  PassengerList(Passenger(114, 'Earl Benton', '23A'),
```
In the example below, you fetch rows from database table `flights` into record `flight_info`. That way, you can treat all the information about a flight, including its passenger list, as a logical unit.

```sql
DECLARE
  TYPE FlightRec IS RECORD (  
    flight_no  NUMBER(3),
    gate       CHAR(5),
    departure  CHAR(15),
    arrival    CHAR(15),
    passengers PassengerList);
  flight_info FlightRec;
  CURSOR c1 IS SELECT * FROM flights;
  seat_not_available EXCEPTION;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO flight_info;
    EXIT WHEN c1%NOTFOUND;
    FOR i IN 1..flight_info.passengers.LAST LOOP
      IF flight_info.passengers(i).seat = 'NA' THEN
        dbms_output.put_line(flight_info.passengers(i).name);
        RAISE seat_not_available;
      END IF;
      ...
    END LOOP;
  END LOOP;
  CLOSE c1;
EXCEPTION
  WHEN seat_not_available THEN
    ...
END;
```

```sql
INSERT INTO flights
  VALUES(27, '34', 'JFK 7:05AM', 'MIA 9:55AM',
    PassengerList(Passenger(27, 'Raymond Kiley', '34D'),
    Passenger(27, 'Beth Steinberg', '3A'),
    Passenger(27, 'Jean Lafevre', '19C'), ...));
```
This chapter helps you harness the power of Oracle. You learn how PL/SQL supports the SQL commands, functions, and operators that let you manipulate Oracle data. You also learn how to manage cursors, use cursor variables, and process transactions.

This chapter discusses the following topics:

- Overview of SQL Support in PL/SQL
- Managing Cursors
- Separating Cursor Specs and Bodies with Packages
- Using Cursor FOR Loops
- Using Cursor Variables
- Using Cursor Attributes
- Using Cursor Expressions
- Overview of Transaction Processing in PL/SQL
- Doing Independent Units of Work with Autonomous Transactions
- Ensuring Backward Compatibility of PL/SQL Programs
Overview of SQL Support in PL/SQL

By extending SQL, PL/SQL offers a unique combination of power and ease of use. You can manipulate Oracle data flexibly and safely because PL/SQL fully supports all SQL data manipulation statements (except EXPLAIN PLAN), transaction control statements, functions, pseudocolumns, and operators. PL/SQL also supports dynamic SQL, which enables you to execute SQL data definition, data control, and session control statements dynamically. In addition, PL/SQL conforms to the current ANSI/ISO SQL standard.

Data Manipulation

To manipulate Oracle data, you use the INSERT, UPDATE, DELETE, SELECT, and LOCK TABLE commands. INSERT adds new rows of data to database tables; UPDATE modifies rows; DELETE removes unwanted rows; SELECT retrieves rows that meet your search criteria; and LOCK TABLE temporarily limits access to a table.

Transaction Control

Oracle is transaction oriented; that is, Oracle uses transactions to ensure data integrity. A transaction is a series of SQL data manipulation statements that does a logical unit of work. For example, two UPDATE statements might credit one bank account and debit another.

Simultaneously, Oracle makes permanent or undoes all database changes made by a transaction. If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction. Thus, the database is restored to its former state automatically.

You use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION commands 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.

SQL Functions

PL/SQL lets you use all the SQL functions including the following aggregate functions, which summarize entire columns of Oracle data: AVG, COUNT, GROUPING, MAX, MIN, STDDEV, SUM, and VARIANCE. Except for COUNT (*), all aggregate functions ignore nulls.
You can use the aggregate functions in SQL statements, but *not* in procedural statements. Aggregate functions operate on entire columns unless you use the `SELECT GROUP BY` statement to sort returned rows into subgroups. If you omit the `GROUP BY` clause, the aggregate function treats all returned rows as a single group.

You call an aggregate function using the syntax

```sql
function_name([ALL | DISTINCT] expression)
```

where `expression` refers to one or more database columns. If you specify `ALL` (the default), the aggregate function considers all column values including duplicates. If you specify `DISTINCT`, the aggregate function considers only distinct values. For example, the following statement returns the number of different job titles in the database table `emp`:

```sql
SELECT COUNT(DISTINCT job) INTO job_count FROM emp;
```

The function `COUNT` lets you use the asterisk (*) row operator, which returns the number of rows in a table. For example, the following statement returns the number of rows in table `emp`:

```sql
SELECT COUNT(*) INTO emp_count FROM emp;
```

### SQL Pseudocolumns

PL/SQL recognizes the following SQL pseudocolumns, which return specific data items: `CURRVAL`, `LEVEL`, `NEXTVAL`, `ROWID`, and `ROWNUM`. Pseudocolumns are not actual columns in a table but they behave like columns. For example, you can select values from a pseudocolumn. However, you cannot insert into, update, or delete from a pseudocolumn. Also, pseudocolumns are allowed in SQL statements, but *not* in procedural statements.

#### 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 obtain the current or next value in a sequence, you must use dot notation, as follows:

```sql
sequence_name.CURRVAL
sequence_name.NEXTVAL
```
After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. However, you can use CURRVAL and NEXTVAL only in a SELECT list, the VALUES clause, and the SET clause. In the following example, you use a sequence to insert the same employee number into two tables:

```sql
INSERT INTO emp VALUES (empno_seq.NEXTVAL, my_ename, ...);
INSERT INTO sals VALUES (empno_seq.CURRVAL, my_sal, ...);
```

If a transaction generates a sequence number, the sequence is incremented immediately whether you commit or roll back the transaction.

**LEVEL**

You use LEVEL with the SELECT CONNECT BY statement to organize rows from a database table into a tree structure. 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 walks 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. In the following example, you declare a variable named row_id for that purpose:

```sql
DECLARE
    row_id UROWID;
```

When you select or fetch a physical rowid into a UROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to an 18-byte character string. Then, 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 6-51.

**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.
You can use ROWNUM in an UPDATE statement to assign unique values to each row in a table. Also, you can use ROWNUM in the WHERE clause of a SELECT statement to limit the number of rows retrieved, as follows:

```plsql
DECLARE
    CURSOR c1 IS SELECT empno, sal FROM emp
    WHERE sal > 2000 AND ROWNUM < 10;  -- returns 10 rows
END;
```

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;
```

### 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 Oracle9i SQL Reference.

#### 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 always yield TRUE, FALSE, or NULL. You can use all the comparison operators listed below to form predicates. Moreover, you can combine predicates using the logical operators AND, OR, and NOT.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Compares a value to each value in a list or returned by a subquery and yields TRUE if all of the individual comparisons yield TRUE.</td>
</tr>
<tr>
<td>ANY, SOME</td>
<td>Compares a value to each value in a list or returned by a subquery and yields TRUE if any of the individual comparisons yields TRUE.</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Tests whether a value lies in a specified range.</td>
</tr>
<tr>
<td>EXISTS</td>
<td>Returns TRUE if a subquery returns at least one row.</td>
</tr>
<tr>
<td>IN</td>
<td>Tests for set membership.</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Tests for nulls.</td>
</tr>
<tr>
<td>LIKE</td>
<td>Tests whether a character string matches a specified pattern, which can include wildcards.</td>
</tr>
</tbody>
</table>
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
PL/SQL uses two types of cursors: **implicit** and **explicit**. PL/SQL declares a cursor implicitly for all SQL data manipulation statements, including queries that return only one row. However, for queries that return more than one row, you must declare an explicit cursor, use a cursor **FOR** loop, or use the **BULK COLLECT** clause.

Overview of Explicit Cursors
The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly declare a cursor to process the rows. Moreover, you can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.

You use three commands 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. You can process several queries in parallel by declaring and opening multiple cursors.

Declaring a Cursor
Forward references are not allowed in PL/SQL. So, you must declare a cursor before referencing it in other statements. When you declare a cursor, you name it and associate it with a specific query using the syntax

```
CURSOR cursor_name [(parameter[, parameter]...)]
```
[RETURN return_type] IS select_statement;

where return_type must represent a record or a row in a database table, and

parameter stands for the following syntax:

cursor_parameter_name [IN] datatype [{:= | DEFAULT} expression]

For example, you might declare cursors named c1 and c2, as follows:

DECLARE
  CURSOR c1 IS SELECT empno, ename, job, sal FROM emp
  WHERE sal > 2000;
  CURSOR c2 RETURN dept%ROWTYPE IS
    SELECT * FROM dept WHERE deptno = 10;

The cursor name is an undeclared identifier, not the name of a PL/SQL variable.
You cannot assign values to a cursor name or use it in an expression. However,
cursors and variables follow the same scoping rules. Naming cursors after database

As the example below shows, you can initialize cursor parameters to default values.
That way, 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 every reference to the cursor.

DECLARE
  CURSOR c1 (low INTEGER DEFAULT 0,
               high INTEGER DEFAULT 99) IS SELECT ...

The scope of cursor parameters is local to the cursor, meaning that they can be
referenced only within the query specified in the cursor declaration. The values of
cursor parameters are used by the associated query when the cursor is opened.

Opening a Cursor

Opening the cursor executes the query and identifies the result set, which consists
of all rows that meet the query search criteria. For cursors declared using the FOR
UPDATE clause, the OPEN statement also locks those rows. An example of the OPEN
statement follows:

DECLARE
CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;
...
BEGIN
  OPEN c1;
  ...
END;

Rows in the result set are not retrieved when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.

**Passing Cursor Parameters**

You use the OPEN statement to pass parameters to a cursor. Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. For example, given the cursor declaration

```plsql
DECLARE
  emp_name emp.ename%TYPE;
  salary   emp.sal%TYPE;
  CURSOR c1 (name VARCHAR2, salary NUMBER) IS SELECT ...;
```

any of the following statements opens the cursor:

```sql
OPEN c1(emp_name, 3000);
OPEN c1(‘ATTLEY’, 1500);
OPEN c1(emp_name, salary);
```

In the last example, when the identifier salary is used in the cursor declaration, it refers to the formal parameter. But, when it is used in the OPEN statement, it refers to the PL/SQL variable. To avoid confusion, use unique identifiers.

Formal parameters declared with a default value need not have a corresponding actual parameter. They can simply 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. The datatypes of each actual parameter and its corresponding formal parameter must be compatible.
Fetching with a Cursor

Unless you use the BULK COLLECT clause (discussed in the next section), the FETCH statement retrieves the rows in the result set one at a time. After each fetch, the cursor advances to the next row in the result set. An example follows:

```sql
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

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 in the following way:

```sql
LOOP
  FETCH c1 INTO my_record;
  EXIT WHEN c1%NOTFOUND;
  -- process data record
END LOOP;
```

The query can reference PL/SQL variables within its scope. However, any variables in the query are evaluated only when the cursor is opened. In the following example, each retrieved salary is multiplied by 2, even though `factor` is incremented after every fetch:

```sql
DECLARE
  my_sal emp.sal%TYPE;
  my_job emp.job%TYPE;
  factor INTEGER := 2;
  CURSOR c1 IS SELECT factor*sal FROM emp WHERE job = my_job;
BEGIN
  ...
  OPEN c1;  -- here factor equals 2
  LOOP
    FETCH c1 INTO my_sal;
    EXIT WHEN c1%NOTFOUND;
    factor := factor + 1;  -- does not affect FETCH
  END LOOP;
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 the following example shows:

```sql
DECLARE
```

```sql
```
Managing Cursors

CURSOR c1 IS SELECT ename FROM emp;

name1 emp.ename%TYPE;
name2 emp.ename%TYPE;
name3 emp.ename%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 indeterminate.

Note: Eventually, the FETCH statement must fail to return a row, so when that happens, no exception is raised. To detect the failure, you must use the cursor attribute %FOUND or %NOTFOUND. For more information, see "Using Cursor Attributes" on page 6-34

Fetching Bulk Data with a Cursor

The BULK COLLECT clause lets you bulk-bind entire columns of Oracle data (see "Retrieving Query Results into Collections with the BULK COLLECT Clause" on page 5-41). That way, you can fetch all rows from the result set at once. In the following example, you bulk-fetch from a cursor into two collections:

DECLARE
  TYPE NumTab IS TABLE OF emp.empno%TYPE;
  TYPE NameTab IS TABLE OF emp.ename%TYPE;
  nums NumTab;
  names NameTab;
  CURSOR c1 IS SELECT empno, ename FROM emp WHERE job = 'CLERK';
BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO nums, names;
  ...
  CLOSE c1;
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. Any other operation on a closed cursor raises the predefined exception INVALID_CURSOR.

Using Subqueries in Cursors

A subquery is a query (usually enclosed by parentheses) that appears within another SQL data manipulation statement. When evaluated, the subquery provides a value or set of values to the statement. Often, subqueries are used in the WHERE clause. For example, the following query returns employees not located in Chicago:

```sql
DECLARE
    CURSOR c1 IS SELECT empno, ename FROM emp
    WHERE deptno IN (SELECT deptno FROM dept
        WHERE loc <> 'CHICAGO');
```

Using a subquery in the FROM clause, the following query returns the number and name of each department with five or more employees:

```sql
DECLARE
    CURSOR c1 IS SELECT t1.deptno, dname, "STAFF"
        FROM dept t1, (SELECT deptno, COUNT(*) "STAFF"
            FROM emp GROUP BY deptno) t2
        WHERE t1.deptno = t2.deptno AND "STAFF" >= 5;
```

Whereas a subquery is evaluated only once for each table, a correlated subquery is evaluated once for each row. Consider the query below, which returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the emp table, the correlated subquery computes the average salary for that row’s department. The row is returned if that row’s salary exceeds the average.

```sql
DECLARE
    CURSOR c1 IS SELECT deptno, ename, sal FROM emp t
    WHERE sal > (SELECT AVG(sal) FROM emp WHERE t.deptno = deptno)
    ORDER BY deptno;
```

Overview of Implicit Cursors

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. You can refer to the most recent implicit cursor as the SQL cursor. Although you cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor, you can use cursor attributes to get information about the most recently executed SQL statement. See “Using Cursor Attributes” on page 6-34.
Separating Cursor Specs and Bodies with Packages

You can separate a cursor specification (spec for short) from its body for placement in a package. That way, you can change the cursor body without having to change the cursor spec. You code the cursor spec in the package spec using this syntax:

```
CURSOR cursor_name [(parameter[, parameter]...)] RETURN return_type;
```

In the following example, you use the %ROWTYPE attribute to provide a record type that represents a row in the database table `emp`:

```
CREATE PACKAGE emp_stuff AS
    CURSOR c1 RETURN emp%ROWTYPE;  -- declare cursor spec
    ...
END emp_stuff;
```

```
CREATE PACKAGE BODY emp_stuff AS
    CURSOR c1 RETURN emp%ROWTYPE IS
        SELECT * FROM emp WHERE sal > 2500;  -- define cursor body
    ...
END emp_stuff;
```

The cursor spec has no `SELECT` statement because the `RETURN` clause specifies the datatype of the return value. However, the cursor body must have a `SELECT` statement and the same `RETURN` clause as the cursor spec. Also, the number and datatypes of items in the `SELECT` list and the `RETURN` clause must match.

Packaged cursors increase flexibility. For instance, you can change the cursor body in the last example, as follows, without having to change the cursor spec:

```
CREATE PACKAGE BODY emp_stuff AS
    CURSOR c1 RETURN emp%ROWTYPE IS
        SELECT * FROM emp WHERE deptno = 20;  -- new WHERE clause
    ...
END emp_stuff;
```

From a PL/SQL block or subprogram, you use dot notation to reference a packaged cursor, as the following example shows:

```
DECLARE
    emp_rec emp%ROWTYPE;
...
BEGIN
    ...
    OPEN emp_stuff.c1;
```

Using Cursor FOR Loops

In most situations that require an explicit cursor, you can simplify coding by using a cursor FOR loop instead of the OPEN, FETCH, and CLOSE statements. 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 have been processed.

Consider the PL/SQL block below, which computes results from an experiment, then stores the results in a temporary table. The FOR loop index c1_rec is implicitly declared as a record. Its fields store all the column values fetched from the cursor c1. Dot notation is used to reference individual fields.

```plsql
-- available online in file 'examp7'
DECLARE
    result temp.col1%TYPE;
    CURSOR c1 IS
        SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
    FOR c1_rec IN c1 LOOP
        /* calculate and store the results */
        result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
        INSERT INTO temp VALUES (result, NULL, NULL);
    END LOOP;
    COMMIT;
END;
```

When the cursor FOR loop is entered, the cursor name cannot belong to a cursor already opened by an OPEN statement or enclosing cursor FOR loop. Before each iteration of the FOR loop, PL/SQL fetches into the implicitly declared record. The record is defined only inside the loop. You cannot refer to its fields outside the loop.
Using Cursor FOR Loops

The sequence of statements inside the loop is executed once for each row that satisfies the query associated with the cursor. When you leave the loop, the cursor is closed automatically—even if you use an EXIT or GOTO statement to leave the loop prematurely or an exception is raised inside the loop.

Using Subqueries Instead of Explicit Cursors

You need not declare a cursor because PL/SQL lets you substitute a subquery. The following cursor FOR loop calculates a bonus, then inserts the result into a database table:

```plsql
DECLARE
    bonus REAL;
BEGIN
    FOR emp_rec IN (SELECT empno, sal, comm FROM emp) LOOP
        bonus := (emp_rec.sal * 0.05) + (emp_rec.comm * 0.25);
        INSERT INTO bonuses VALUES (emp_rec.empno, bonus);
    END LOOP;
    COMMIT;
END;
```

Using Cursor Subqueries

You can use cursor subqueries, also known as cursor expressions, to pass sets of rows as parameters to functions. For example, this statement passes a parameter to the StockPivot function consisting of a REF CURSOR that represents the rows returned by the cursor subquery:

```plsql
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

Cursor subqueries are often used with table functions, which are explained in "Accepting and Returning Multiple Rows with Table Functions" on page 8-30.

Defining Aliases for Expression Values in a Cursor FOR Loop

Fields in the implicitly declared record hold column values from the most recently fetched row. The fields have the same names as corresponding columns in the SELECT list. But, what happens if a select item is an expression? Consider the following example:

```plsql
CURSOR c1 IS
    SELECT empno, sal+NVL(comm,0), job FROM ...
In such cases, you must include an alias for the select item. In the following example, wages is an alias for the select item sal+NVL(comm,0):

```sql
CURSOR c1 IS
  SELECT empno, sal+NVL(comm,0) wages, job FROM ...;
```

To reference the corresponding field, use the alias instead of a column name, as follows:

```sql
IF emp_rec.wages < 1000 THEN ...;
```

### Passing Parameters to a Cursor FOR Loop

You can pass parameters to the cursor in a cursor FOR loop. In the following example, you pass a department number. Then, you compute the total wages paid to employees in that department. Also, you determine how many employees have salaries higher than $2000 and/or commissions larger than their salaries.

```sql
DECLARE
  CURSOR emp_cursor(dnum NUMBER) IS
    SELECT sal, comm FROM emp WHERE deptno = dnum;
  total_wages NUMBER(11,2) := 0;
  high_paid   NUMBER(4) := 0;
  higher_comm NUMBER(4) := 0;
BEGIN
  /* The number of iterations will equal the number of rows
  returned by emp_cursor. */
  FOR emp_record IN emp_cursor(20) LOOP
    emp_record.comm := NVL(emp_record.comm, 0);
    total_wages := total_wages + emp_record.sal +
      emp_record.comm;
    IF emp_record.sal > 2000.00 THEN
      high_paid := high_paid + 1;
    END IF;
    IF emp_record.comm > emp_record.sal THEN
      higher_comm := higher_comm + 1;
    END IF;
  END LOOP;
  INSERT INTO temp VALUES (high_paid, higher_comm,
    'Total Wages: ' || TO_CHAR(total_wages));
  COMMIT;
END;
```
Using Cursor Variables

Like a cursor, a cursor variable points to the current row in the result set of a multi-row query. But, cursors differ from cursor variables the way constants differ from variables. Whereas a cursor is static, a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query. This gives you more flexibility.

Also, you can assign new values to a cursor variable and pass it as a parameter to local and stored subprograms. This gives you an easy way to centralize data retrieval.

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. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side.

The Oracle server also has a PL/SQL engine. So, you can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs).

What Are Cursor Variables?

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, not an item. In PL/SQL, a pointer has datatype REF X, where REF is short for REFERENCE and X stands for a class of objects. Therefore, a cursor variable has datatype REF CURSOR.

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use an explicit cursor, which names the work area. Or, you can use a cursor variable, which points to the work area. Whereas a cursor always refers to the same query work area, a cursor variable can refer to different work areas. So, cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected.

Why Use Cursor Variables?

Mainly, you use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle server can all refer to the same work area.
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.

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. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single round trip.

Defining REF CURSOR Types

To create cursor variables, you take two steps. First, 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 using the syntax

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

where `ref_type_name` is a type specifier used in subsequent declarations of cursor variables and `return_type` must represent a record or a row in a database table. In the following example, you specify a return type that represents a row in the database table `dept`:

```
DECLARE
  TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
```

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). As the next example shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;  -- strong
  TYPE GenericCurTyp IS REF CURSOR;  -- weak
```

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.
Declaring Cursor Variables

Once you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable dept_cv:

```sql
DECLARE
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
    dept_cv DeptCurTyp;  -- declare cursor variable
```

**Note:** You cannot declare cursor variables in a package. Unlike packaged variables, cursor variables do not have persistent state. Remember, declaring a cursor variable creates a pointer, not an item. So, cursor variables cannot be saved in the database.

Cursor variables follow the usual scoping and instantiation rules. Local PL/SQL cursor variables are instantiated when you enter a block or subprogram and cease to exist when you exit.

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, as follows:

```sql
DECLARE
    TYPE TmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    tmp_cv TmpCurTyp;  -- declare cursor variable
    TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
    emp_cv EmpCurTyp;  -- declare cursor variable
```

Likewise, you can use %TYPE to provide the datatype of a record variable, as the following example shows:

```sql
DECLARE
    dept_rec dept%ROWTYPE;  -- declare record variable
    TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
    dept_cv DeptCurTyp;  -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```sql
DECLARE
    TYPE EmpRecTyp IS RECORD (
        empno NUMBER(4),
        ename VARCHAR2(10),
        sal   NUMBER(7,2));
    TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
    emp_cv EmpCurTyp;  -- declare cursor variable
```
Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following example, you define the REF CURSOR type EmpCurTyp, then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ... 
```

Caution: Like all pointers, cursor variables increase the possibility of parameter aliasing. See "Understanding Subprogram Parameter Aliasing" on page 8-21.

Controlling Cursor Variables

You use three statements to control a cursor variable: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set. When all the rows are processed, you CLOSE the cursor variable.

Opening a Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi-row query, executes the query, and identifies the result set. Here is the syntax:

```
OPEN {cursor_variable | :host_cursor_variable} FOR 
{ select_statement 
  | dynamic_string [USING bind_argument[, bind_argument]...] };
```

where host_cursor_variable is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multi-row query.

Note: This section discusses the static SQL case, in which select_statement is used. For the dynamic SQL case, in which dynamic_string is used, see "Opening the Cursor Variable" on page 11-8.

Unlike cursors, cursor variables take no parameters. However, no flexibility is lost because 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.
In the example below, you open the cursor variable `emp_cv`. Notice that you can apply cursor attributes (%FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT) to a cursor variable.

```plsql
IF NOT emp_cv%ISOPEN THEN
  /* Open cursor variable. */
  OPEN emp_cv FOR SELECT * FROM emp;
END IF;
```

Other `OPEN-FOR` statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. (Recall that consecutive `OPENs` of a static cursor raise the predefined exception CURSOR_ALREADY_OPEN.) When you reopen a cursor variable for a different query, the previous query is lost.

Typically, you open a cursor variable by passing it to a stored procedure that declares a cursor variable as one of its formal parameters. For example, the following packaged procedure opens the cursor variable `emp_cv`:

```sql
CREATE PACKAGE emp_data AS
  ...
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%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 emp;
    END open_emp_cv;
END emp_data;
```

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.

Alternatively, you can use a standalone procedure to open the cursor variable. Simply define the `REF CURSOR` type in a separate package, then reference that type in the standalone procedure. For instance, if you create the following bodiless package, you can create standalone procedures that reference the types it defines:

```sql
CREATE PACKAGE cv_types AS
  TYPE GenericCurTyp IS REF CURSOR;
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
  ...
```
In the next example, you create a standalone procedure that references the `REF CURSOR` type `EmpCurTyp`, which is defined in the package `cv_types`:

```sql
CREATE PROCEDURE open_emp_cv (emp_cv IN OUT cv_types.EmpCurTyp) AS
BEGIN
  OPEN emp_cv FOR SELECT * FROM emp;
END open_emp_cv;
```

To centralize data retrieval, you can group type-compatible queries in a stored procedure. In the example below, the packaged procedure declares a selector as one of its formal parameters. When called, the procedure opens the cursor variable `emp_cv` for the chosen query.

```sql
CREATE PACKAGE emp_data AS
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice INT);
END emp_data;
```

```sql
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 emp WHERE comm IS NOT NULL;
    ELSIF choice = 2 THEN
      OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
    ELSIF choice = 3 THEN
      OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
    END IF;
  END;
END emp_data;
```

For more flexibility, you can pass a cursor variable and a selector to a stored procedure that executes queries with different return types. Here is an example:

```sql
CREATE PACKAGE admin_data AS
  TYPE GenCurTyp IS REF CURSOR;
  PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT);
END admin_data;
```

```sql
CREATE PACKAGE BODY admin_data AS
  PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
    END;
END admin_data;
```

```sql
END cv_types;
```
Using Cursor Variables

OPEN generic_cv FOR SELECT * FROM emp;
ELSIF choice = 2 THEN
  OPEN generic_cv FOR SELECT * FROM dept;
ELSIF choice = 3 THEN
  OPEN generic_cv FOR SELECT * FROM salgrade;
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 SQLALLOCATE :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 emp;
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM dept;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM salgrade;
  END IF;
END;
END-EXEC;

Host cursor variables are compatible with any query return type. They behave just like weakly typed PL/SQL cursor variables.
Fetching from a Cursor Variable

The \texttt{FETCH} statement retrieves rows from the result set of a multi-row query. Here is the syntax:

\begin{verbatim}
FETCH \{cursor\_variable\_name | :host\_cursor\_variable\_name\}
[BULK COLLECT]
INTO \{variable\_name[, variable\_name]... | record\_name\};
\end{verbatim}

In the following example, you fetch rows one at a time from the cursor variable \texttt{emp\_cv} into the user-defined record \texttt{emp\_rec}:

\begin{verbatim}
LOOP
  /* Fetch from cursor variable. */
  FETCH emp_cv INTO emp_rec;
  EXIT WHEN emp_cv\%NOTFOUND; -- exit when last row is fetched
  -- process data record
END LOOP;
\end{verbatim}

Using the \texttt{BULK COLLECT} clause, you can bulk fetch rows from a cursor variable into one or more collections. An example follows:

\begin{verbatim}
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp\%ROWTYPE;
  TYPE NameList IS TABLE OF emp.ename\%TYPE;
  TYPE SalList IS TABLE OF emp.sal\%TYPE;
  emp_cv EmpCurTyp;
  names NameList;
  sals SalList;
BEGIN
  OPEN emp_cv FOR SELECT ename, sal FROM emp;
  FETCH emp_cv BULK COLLECT INTO names, sals;
  ...
END;
\end{verbatim}

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, you must reopen the cursor variable with the variables set to their new values. However, you can use a different \texttt{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. 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. The 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. So, if you trap the error and execute the FETCH statement using a different 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. However, 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. After that, the associated result set is undefined. Here is the syntax:

```
CLOSE {cursor_variable_name | :host_cursor_variable_name);
```

In the following example, when the last row is processed, you close the cursor variable `emp_cv`:

```
LOOP
    FETCH emp_cv INTO emp_rec;
    EXIT WHEN emp_cv%NOTFOUND;
    -- process data record
END LOOP;
/* Close cursor variable. */
CLOSE emp_cv;
```

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.
Cursor Variable Example: Master Table and Details Tables

Consider the stored procedure below, which searches the database of a main library for books, periodicals, and tapes. A master table stores the title and category code (where 1 = book, 2 = periodical, 3 = tape) of each item. Three detail tables store category-specific information. When called, the procedure searches the master table by title, uses the associated category code to pick an OPEN-FOR statement, then opens a cursor variable for a query of the proper detail table.

```sql
CREATE PACKAGE cv_types AS
    TYPE LibCurTyp IS REF CURSOR;
    ...
END cv_types;

CREATE PROCEDURE find_item (
    title VARCHAR2(100),
    lib_cv IN OUT cv_types.LibCurTyp)
AS
    code BINARY_INTEGER;
BEGIN
    SELECT item_code FROM titles INTO code
    WHERE item_title = title;
    IF code = 1 THEN
        OPEN lib_cv FOR SELECT * FROM books
        WHERE book_title = title;
    ELSIF code = 2 THEN
        OPEN lib_cv FOR SELECT * FROM periodicals
        WHERE periodical_title = title;
    ELSIF code = 3 THEN
        OPEN lib_cv FOR SELECT * FROM tapes
        WHERE tape_title = title;
    END IF;
END find_item;
```
Cursor Variable Example: Client-Side PL/SQL Block

A client-side application in a branch library might use the following PL/SQL block to display the retrieved information:

```plsql
DECLARE
    lib_cv         cv_types.LibCurTyp;
    book_rec       books%ROWTYPE;
    periodical_rec periodicals%ROWTYPE;
    tape_rec       tapes%ROWTYPE;
BEGIN
    get_title(:title);  -- title is a host variable
    find_item(:title, lib_cv);
    FETCH lib_cv INTO book_rec;
    display_book(book_rec);
EXCEPTION
    WHEN ROWTYPE_MISMATCH THEN
        BEGIN
            FETCH lib_cv INTO periodical_rec;
            display_periodical(periodical_rec);
        EXCEPTION
            WHEN ROWTYPE_MISMATCH THEN
                FETCH lib_cv INTO tape_rec;
                display_tape(tape_rec);
        END;
END;
```

Cursor Variable Example: Pro*C Program

The following Pro*C program prompts the user to select a database table, opens a cursor variable for a query of that table, then fetches rows returned by the query:

```c
#include <stdio.h>
#include <sqlca.h>

void sql_error();

main()
{
    char temp[32];

    EXEC SQL BEGIN DECLARE SECTION;

    char * uid = "scott/tiger";
    SQL_CURSOR generic_cv; /* cursor variable */
    int    table_num;     /* selector */
    struct /* EMP record */
```
Using Cursor Variables

```c

struct                   /* DEPT record */
{
  int dept_num;
  char dept_name[15];
  char location[14];
} dept_rec;

struct                   /* BONUS record */
{
  char emp_name[11];
  char job_title[10];
  float salary;
} bonus_rec;

EXEC SQL END DECLARE SECTION;

/* Handle Oracle errors. */
EXEC SQL WHENEVER SQLERROR DO sql_error();

/* Connect to Oracle. */
EXEC SQL CONNECT :uid;

/* Initialize cursor variable. */
EXEC SQL ALLOCATE :generic_cv;

/* Exit loop when done fetching. */
EXEC SQL WHENEVER NOT FOUND DO break;

for (;;)
{
  printf("\n1 = EMP, 2 = DEPT, 3 = BONUS\n");
  printf("\nEnter table number (0 to quit): ");
  gets(temp);
  table_num = atoi(temp);
  if (table_num <= 0) break;
```
Using Cursor Variables

```sql
/* Open cursor variable. */
EXEC SQL EXECUTE
BEGIN
    IF :table_num = 1 THEN
        OPEN :generic_cv FOR SELECT * FROM emp;
    ELSIF :table_num = 2 THEN
        OPEN :generic_cv FOR SELECT * FROM dept;
    ELSIF :table_num = 3 THEN
        OPEN :generic_cv FOR SELECT * FROM bonus;
    END IF;
END;
END-EXEC;

for (;;)
{
    switch (table_num)
    {
        case 1: /* Fetch row into EMP record. */
            EXEC SQL FETCH :generic_cv INTO :emp_rec;
            break;
        case 2: /* Fetch row into DEPT record. */
            EXEC SQL FETCH :generic_cv INTO :dept_rec;
            break;
        case 3: /* Fetch row into BONUS record. */
            EXEC SQL FETCH :generic_cv INTO :bonus_rec;
            break;
    }
    /* Process data record here. */
}
/* Close cursor variable. */
EXEC SQL CLOSE :generic_cv;
exit(0);
}
void sql_error()
{
    /* Handle SQL error here. */
}
```
Cursor Variable Example: Manipulating Host Variables in SQL*Plus

A host variable is a variable you declare in a host environment, then pass to one or more PL/SQL programs, which can use it like any other variable. In the SQL*Plus environment, to declare a host variable, use the command VARIABLE. For example, you declare a variable of type NUMBER as follows:

```sql
VARIABLE return_code NUMBER
```

Both SQL*Plus and PL/SQL can reference the host variable, and SQL*Plus can display its value. However, to reference a host variable in PL/SQL, you must prefix its name with a colon (:), as the following example shows:

```sql
DECLARE
    ...
BEGIN
    :return_code := 0;
    IF credit_check_ok(acct_no) THEN
        :return_code := 1;
    END IF;
    ...
END;
```

To display the value of a host variable in SQL*Plus, use the PRINT command, as follows:

```sql
SQL> PRINT return_code
RETURN_CODE
----------
     1
```

The SQL*Plus datatype REFCURSOR lets you declare cursor variables, which you can use to return query results from stored subprograms. In the script below, you declare a host variable of type REFCURSOR. You use the SQL*Plus command SET AUTOPRINT ON to display the query results automatically.

```sql
CREATE PACKAGE emp_data AS
    TYPE EmpRecTyp IS RECORD (emp_id NUMBER(4), emp_name VARCHAR2(10), job_title VARCHAR2(9), dept_name VARCHAR2(14), dept_loc VARCHAR2(13));
    TYPE EmpCurTyp IS REFCURSOR RETURN EmpRecTyp;
    PROCEDURE get_staff (}
Creating a package body for the `emp_data` package:

```sql
CREATE PACKAGE BODY emp_data AS
PROCEDURE get_staff ( 
    dept_no IN NUMBER, 
    emp_cv IN OUT EmpCurTyp) IS
BEGIN 
    OPEN emp_cv FOR 
        SELECT empno, ename, job, dname, loc FROM emp, dept 
        WHERE emp.deptno = dept_no AND emp.deptno = dept.deptno 
        ORDER BY empno;
END;
END;
/
```

**Reduction of 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 five cursor variables in a single round trip:

```sql
/* anonymous PL/SQL block in host environment */
BEGIN 
    OPEN :emp_cv FOR SELECT * FROM emp; 
    OPEN :dept_cv FOR SELECT * FROM dept; 
    OPEN :grade_cv FOR SELECT * FROM salgrade; 
    OPEN :pay_cv FOR SELECT * FROM payroll; 
    OPEN :ins_cv FOR SELECT * FROM insurance; 
END;
```

This might be useful in Oracle Forms, for instance, when you want to populate a multi-block form.
When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes. That allows your OCI or Pro*C program to use these work areas for ordinary cursor operations. In the following example, you open several such work areas in a single round trip:

```plsql
BEGIN
  OPEN :c1 FOR SELECT 1 FROM dual;
  OPEN :c2 FOR SELECT 1 FROM dual;
  OPEN :c3 FOR SELECT 1 FROM dual;
  OPEN :c4 FOR SELECT 1 FROM dual;
  OPEN :c5 FOR SELECT 1 FROM dual;
  ...
END;
```

The cursors assigned to c1, c2, c3, c4, and c5 behave normally, and you can use them for any purpose. When finished, simply release the cursors, as follows:

```plsql
BEGIN
  CLOSE :c1;
  CLOSE :c2;
  CLOSE :c3;
  CLOSE :c4;
  CLOSE :c5;
  ...
END;
```

### Avoiding Errors with Cursor Variables

If both cursor variables involved in an assignment are strongly typed, they must have the same datatype. In the following example, even though the cursor variables have the same return type, the assignment raises an exception because they have different datatypes:

```plsql
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  TYPE TmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, tmp_cv IN OUT TmpCurTyp) IS
BEGIN
  ...
  emp_cv := tmp_cv;  -- causes 'wrong type' error
END;
```
However, if one or both cursor variables are weakly typed, they need not have the same datatype.

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 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.

The following example shows how these ways interact:

```sql
DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  emp_cv1 EmpCurTyp;
  emp_cv2 EmpCurTyp;
  emp_rec emp%ROWTYPE;
BEGIN
  /* The following assignment is useless because emp_cv1
     does not point to a query work area yet. */
  emp_cv2 := emp_cv1;  -- useless
  /* Make emp_cv1 point to a query work area. */
  OPEN emp_cv1 FOR SELECT * FROM emp;
  /* Use emp_cv1 to fetch first row from emp table. */
  FETCH emp_cv1 INTO emp_rec;
  /* The following fetch raises an exception because emp_cv2
     does not point to a query work area yet. */
  FETCH emp_cv2 INTO emp_rec;  -- raises INVALID_CURSOR
EXCEPTION
  WHEN INVALID_CURSOR THEN
    /* Make emp_cv1 and emp_cv2 point to same work area. */
    emp_cv2 := emp_cv1;
    /* Use emp_cv2 to fetch second row from emp table. */
    FETCH emp_cv2 INTO emp_rec;
    /* Reuse work area for another query. */
    OPEN emp_cv2 FOR SELECT * FROM old_emp;
    /* Use emp_cv1 to fetch first row from old_emp table.
     The following fetch succeeds because emp_cv1 and
     emp_cv2 point to the same query work area. */
    FETCH emp_cv1 INTO emp_rec;  -- succeeds
END;
```
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.

In the Pro*C example below, you define a packaged REF CURSOR type, specifying the return type emp%ROWTYPE. Next, you create a standalone procedure that references the new type. Then, inside a PL/SQL block, you open a host cursor variable for a query of the dept table. Later, when you pass the open host cursor variable to the stored procedure, PL/SQL raises ROWTYPE_MISMATCH because the return types of the actual and formal parameters are incompatible.

```sql
CREATE PACKAGE cv_types AS
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    ...
END cv_types;
/
CREATE PROCEDURE open_emp_cv (emp_cv IN OUT cv_types.EmpCurTyp) AS
    BEGIN
        OPEN emp_cv FOR SELECT * FROM emp;
    END open_emp_cv;
/
-- anonymous PL/SQL block in Pro*C program
EXEC SQL EXECUTE
    BEGIN
        OPEN :cv FOR SELECT * FROM dept;
        ...
        open_emp_cv(:cv); -- raises ROWTYPE_MISMATCH
    END;
END-EXEC;
```

**Restrictions on Cursor Variables**

Currently, cursor variables are subject to the following restrictions:

- You cannot declare cursor variables in a package. For example, the following declaration is not allowed:

  ```sql
  CREATE PACKAGE emp_stuff AS
      TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
      emp_cv EmpCurTyp; -- not allowed
  END emp_stuff;
  ```

- Remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use RPCs to pass cursor variables from one server to another.
Using Cursor Attributes

- If you pass a host cursor variable to PL/SQL, 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.
- You cannot assign nulls to a cursor variable.
- You cannot use \texttt{REF CURSOR} types to specify column types in a \texttt{CREATE TABLE} or \texttt{CREATE VIEW} statement. So, database columns cannot store the values of cursor variables.
- You cannot use a \texttt{REF CURSOR} type to specify the element type of a collection, which means that elements in an index-by table, nested table, or varray cannot store the values of cursor variables.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected. For example, the following cursor \texttt{FOR} loop is not allowed because it attempts to fetch from a cursor variable:

  \begin{verbatim}
  DECLARE
    TYPE EmpCurTyp IS REF CURSOR RETURN emp\%ROWTYPE;
    emp_cv EmpCurTyp;
  ...
  BEGIN
    ...
    FOR emp_rec IN emp_cv LOOP ...
    END;
  \end{verbatim}

Using Cursor Attributes

Every explicit cursor and cursor variable has four attributes: \%FOUND, \%ISOPEN \%NOTFOUND, and \%ROWCOUNT. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a data manipulation statement. You can use cursor attributes in procedural statements but not in SQL statements.

Overview of Explicit Cursor Attributes

Explicit cursor attributes return information about the execution of a multi-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.
%FOUND Attribute: Has a Row Been Fetched?
After a cursor or cursor variable is opened but before the first fetch, %FOUND yields NULL. Thereafter, it yields TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row. In the following example, you use %FOUND to select an action:

```
LOOP
    FETCH c1 INTO my_ename, my_sal, my_hiredate;
    IF c1%FOUND THEN  -- fetch succeeded
        ...
    ELSE  -- fetch failed, so exit loop
        EXIT;
    END IF;
END LOOP;
```

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 yields TRUE if its cursor or cursor variable is open; otherwise, %ISOPEN yields FALSE. In the following example, you use %ISOPEN to select an action:

```
IF c1%ISOPEN THEN  -- cursor is open
    ...
ELSE  -- cursor is closed, so open it
    OPEN c1;
END IF;
```

%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 the following example, you use %NOTFOUND to exit a loop when FETCH fails to return a row:

```
LOOP
    FETCH c1 INTO my_ename, my_sal, my_hiredate;
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;
```

Before the first fetch, %NOTFOUND evaluates to NULL. So, if FETCH never executes successfully, the loop is never exited. That is 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 INVALID_CURSOR.

%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 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. In the next example, you use %ROWCOUNT to take action if more than ten rows have been fetched:

```
LOOP
    FETCH c1 INTO my_ename, my_deptno;
    IF c1%ROWCOUNT > 10 THEN
        ...
    END IF;
    ...
END LOOP;
```

If a cursor or cursor variable is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR.

Table 6–1 shows what each cursor attribute yields before and after you execute an OPEN, FETCH, or CLOSE statement.

**Table 6–1  Cursor Attribute Values**

<table>
<thead>
<tr>
<th></th>
<th>%FOUND</th>
<th>%ISOPEN</th>
<th>%NOTFOUND</th>
<th>%ROWCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>before</td>
<td>exception</td>
<td>FALSE</td>
<td>exception</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td>First FETCH</td>
<td>before</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Next FETCH(es)</td>
<td>before</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Last FETCH</td>
<td>before</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>CLOSE</td>
<td>before</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>exception</td>
<td>FALSE</td>
<td>exception</td>
</tr>
</tbody>
</table>
Some Examples of Cursor Attributes

Suppose you have a table named `data_table` that holds data collected from laboratory experiments, and you want to analyze the data from experiment 1. In the following example, you compute the results and store them in a database table named `temp`:

```
-- available online in file 'examp5'
DECLARE
  num1   data_table.n1%TYPE;  -- Declare variables
  num2   data_table.n2%TYPE;  -- having same types as
  num3   data_table.n3%TYPE;  -- database columns
  result temp.col1%TYPE;
CURSOR c1 IS
  SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO num1, num2, num3;
    EXIT WHEN c1%NOTFOUND;  -- TRUE when FETCH finds no more rows
    result := num2/(num1 + num3);
    INSERT INTO temp VALUES (result, NULL, NULL);
  END LOOP;
  CLOSE c1;
  COMMIT;
END;
```

In the next example, you check all storage bins that contain part number 5469, withdrawing their contents until you accumulate 1000 units:

```
-- available online in file 'examp6'
DECLARE
  CURSOR bin_cur(part_number NUMBER) IS
    SELECT amt_in_bin FROM bins
    WHERE part_num = part_number AND amt_in_bin > 0
BEGIN
  LOOP
    FETCH bin_cur INTO part_num, amt_in_bin;
    EXIT WHEN bin_cur%NOTFOUND;  -- TRUE when FETCH finds no more rows
    amt_in_bin :=amt_in_bin - 1000;
    -- do whatever you need to withdraw
  END LOOP;
END;
```

Notes:

1. Referencing %FOUND, %NOTFOUND, or %ROWCOUNT before a cursor is opened or after it is closed raises INVALID_CURSOR.
2. After the first FETCH, if the result set was empty, %FOUND yields FALSE, %NOTFOUND yields TRUE, and %ROWCOUNT yields 0.

### Table 6–1 Cursor Attribute Values

<table>
<thead>
<tr>
<th><code>%FOUND</code></th>
<th><code>%ISOPEN</code></th>
<th><code>%NOTFOUND</code></th>
<th><code>%ROWCOUNT</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ORDER BY bin_num
FOR UPDATE OF amt_in_bin;
bin_amt bins.amt_in_bin%TYPE;
total_so_far NUMBER(5) := 0;
amount_needed CONSTANT NUMBER(5) := 1000;
bins_looked_at NUMBER(3) := 0;
BEGIN
OPEN bin_cur(5469);
WHILE total_so_far < amount_needed LOOP
    FETCH bin_cur INTO bin_amt;
    EXIT WHEN bin_cur%NOTFOUND;
    -- if we exit, there’s not enough to fill the order
    bins_looked_at := bins_looked_at + 1;
    IF total_so_far + bin_amt < amount_needed THEN
        UPDATE bins SET amt_in_bin = 0
        WHERE CURRENT OF bin_cur;
        -- take everything in the bin
        total_so_far := total_so_far + bin_amt;
    ELSE -- we finally have enough
        UPDATE bins SET amt_in_bin = amt_in_bin
        - (amount_needed - total_so_far)
        WHERE CURRENT OF bin_cur;
        total_so_far := amount_needed;
    END IF;
END LOOP;
CLOSE bin_cur;
INSERT INTO temp
VALUES (NULL, bins_looked_at, '<- bins looked at');
COMMIT;
END;

Overview of Implicit Cursor Attributes

Implicit cursor attributes return information about the execution of an INSERT,
UPDATE, DELETE, or SELECT INTO statement. The values of the cursor attributes
always refer to the most recently executed SQL statement. Before Oracle opens the
SQL cursor, the implicit cursor attributes yield NULL.

Note: The SQL cursor has another attribute, %BULK_ROWCOUNT, designed for use
with the FORALL statement. For more information, see “Counting Rows Affected by
FORALL Iterations with the %BULK_ROWCOUNT Attribute” on page 5-36.
Using Cursor Attributes

%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 the following example, you use %FOUND to insert a row if a delete succeeds:

```
DELETE FROM emp WHERE empno = my_empno;
IF SQL%FOUND THEN  -- delete succeeded
    INSERT INTO new_emp VALUES (my_empno, my_ename, ...);
```

%ISOPEN Attribute: Always FALSE for Implicit Cursors
Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE.

%NOTFOUND Attribute: Has a DML Statement Failed to Change Rows?
%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.

%ROWCOUNT Attribute: How Many Rows Affected So Far?
%ROWCOUNT yields the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. %ROWCOUNT yields 0 if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. In the following example, you use %ROWCOUNT to take action if more than ten rows have been deleted:

```
DELETE FROM emp WHERE ...  
IF SQL%ROWCOUNT > 10 THEN  -- more than 10 rows were deleted
    ...
END IF;
```

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.

Guidelines for Using Implicit Cursor Attributes
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). So, if you want to save an attribute value for later use, assign it to a
Boolean variable immediately. In the following example, relying on the IF condition is dangerous because the procedure check_status might have changed the value of %NOTFOUND:

```
BEGIN
  ...
  UPDATE parts SET quantity = quantity - 1 WHERE partno = part_id;
  check_status(part_id); -- procedure call
  IF SQL%NOTFOUND THEN -- dangerous!
    ...
  END;
END;
```

You can improve the code as follows:

```
BEGIN
  ...
  UPDATE parts SET quantity = quantity - 1 WHERE partno = part_id;
  sql_notfound := SQL%NOTFOUND; -- assign value to Boolean variable
  check_status(part_id);
  IF sql_notfound THEN ...
END;
```

If a SELECT INTO statement fails to return a row, PL/SQL raises the predefined exception NO_DATA_FOUND whether you check %NOTFOUND on the next line or not. Consider the following example:

```
BEGIN
  ...
  SELECT sal INTO my_sal FROM emp WHERE empno = my_empno;
  -- might raise NO_DATA_FOUND
  IF SQL%NOTFOUND THEN -- condition tested only when false
    ... -- this action is never taken
  END IF;
```

The check is useless because the IF condition is tested only when %NOTFOUND is false. When PL/SQL raises NO_DATA_FOUND, normal execution stops and control transfers to the exception-handling part of the block.

However, a SELECT INTO statement that calls a SQL aggregate function never raises NO_DATA_FOUND because aggregate functions always return a value or a null. In such cases, %NOTFOUND yields FALSE, as the following example shows:

```
BEGIN
  ...
  SELECT MAX(sal) INTO my_sal FROM emp WHERE deptno = my_deptno;
  -- never raises NO_DATA_FOUND
```
Using Cursor Expressions

A cursor expression returns a nested cursor. Each row in the result set can contain values as usual, plus cursors produced by subqueries involving the other values in the row. Thus, a single query can return a large set of related values retrieved from multiple tables. You can process the result set with nested loops that fetch first from the rows of the result set, then any nested cursors within those rows.

PL/SQL supports queries with cursor expressions as part of cursor declarations, REF CURSOR declarations and ref cursor variables. You can also use cursor expressions in dynamic SQL queries. Here is the syntax:

```
CURSOR ( subquery )
```

A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is 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.

Restrictions on Cursor Expressions

- You cannot use a cursor expression with an implicit cursor.
- Cursor expressions can appear only:
  - In a `SELECT` statement that is not nested in any other query expression, except when it is a subquery of the cursor expression itself.
  - As arguments to table functions, in the `FROM` clause of a `SELECT` statement.
Cursor expressions can appear only in the outermost SELECT list of the query specification.

Cursor expressions cannot appear in view declarations.

You cannot perform BIND and EXECUTE operations on cursor expressions.

**Example of Cursor Expressions**

In this example, we find a specified location ID, and a cursor from which we can fetch all the departments in that location. As we fetch each department’s name, we also get another cursor that lets us fetch their associated employee details from another table.

```sql
CREATE OR REPLACE procedure emp_report(p_locid number) is
  TYPE refcursor is ref cursor;
  -- The query returns only 2 columns, but the second column is
  -- a cursor that lets us traverse a set of related information.
  CURSOR c1 is
    SELECT l.city,
    CURSOR(SELECT d.department_name,
    CURSOR(SELECT e.last_name
    FROM employees e
    WHERE e.department_id = d.department_id) as ename
    FROM departments d
    WHERE l.location_id = d.location_id) dname
    FROM locations l
    WHERE l.location_id = p_locid;

  loccur refcursor;
  deptcur refcursor;
  empcur refcursor;
  V_city locations.city%type;
  V_dname departments.department_name%type;
  V_ename employees.last_name%type;

  BEGIN
    OPEN c1;
    LOOP
      FETCH C1 INTO v_city, loccur;
      EXIT WHEN c1%notfound;
      -- We can access the column C1.city, then process the results of
      -- the nested cursor.
    END LOOP;
  END;
END;
```
Overview of Transaction Processing in PL/SQL

This section explains how to do transaction processing. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to Oracle data are made permanent or undone.

The jobs or tasks that Oracle manages are called sessions. A user session is started when you run an application program or an Oracle tool and connect to Oracle. To allow user sessions to work “simultaneously” and share computer resources, Oracle must control concurrency, the accessing of the same data by many users. Without adequate concurrency controls, there might be a loss of data integrity. That is, changes to data might be made in the wrong order.

Oracle uses locks to control concurrent access to data. A lock gives you temporary ownership of a database resource such as a table or row of data. Thus, data cannot be changed by other users until you finish with it. You need never explicitly lock a resource because default locking mechanisms protect Oracle data and structures. However, you can request data locks on tables or rows when it is to your advantage to override default locking. You can choose from several modes of locking such as row share and exclusive.

A deadlock can occur when two or more users try to access the same schema object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until Oracle breaks the deadlock by signaling an error to the last participating transaction.

```
LOOP
FETCH loccur INTO v_dname, deptcur; -- No need to open
EXIT WHEN loccur%notfound;
LOOP
FETCH deptcur into v_ename; -- No need to open
EXIT WHEN deptcur%notfound;
DBMS_OUTPUT.PUT_LINE(v_city ||' '||v_dname||' '||v_ename);
END LOOP;
END LOOP;
close c1;
END;
/
```
When a table is being queried by one user and updated by another at the same time, Oracle generates a read-consistent view of the data for the query. That is, once a query begins and as it proceeds, the data read by the query does not change. As update activity continues, Oracle takes snapshots of the table’s data and records changes in a rollback segment. Oracle uses rollback segments to build read-consistent query results and to undo changes if necessary.

How Transactions Guard Your Database

A transaction is a series of SQL data manipulation statements that does a logical unit of work. Oracle treats the series of SQL statements as a unit so that all the changes brought about by the statements are either committed (made permanent) or rolled back (undone) at the same time. If your program fails in the middle of a transaction, the database is automatically restored to its former state.

The first SQL statement in your program begins a transaction. When one transaction ends, the next SQL statement automatically begins another transaction. Thus, every SQL statement is part of a transaction. A distributed transaction includes at least one SQL statement that updates data at multiple nodes in a distributed database.

The COMMIT and ROLLBACK statements ensure that all database changes brought about by SQL operations are either made permanent or undone at the same time. All the SQL statements executed since the last commit or rollback make up the current transaction. The SAVEPOINT statement names and marks the current point in the processing of a transaction.

Making Changes Permanent with COMMIT

The COMMIT statement ends the current transaction and makes permanent any changes made during that transaction. Until you commit the changes, other users cannot access the changed data; they see the data as it was before you made the changes.

Consider a simple transaction that transfers money from one bank account to another. The transaction requires two updates because it debits the first account, then credits the second. In the example below, after crediting the second account, you issue a commit, which makes the changes permanent. Only then do other users see the changes.

BEGIN

... UPDATE accts SET bal = my_bal - debit
 WHERE acctno = 7715;

END
The COMMIT statement releases all row and table locks. It also erases any savepoints (discussed later) marked since the last commit or rollback. The optional keyword WORK has no effect other than to improve readability. The keyword END signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.

The optional COMMENT clause lets you specify a comment to be associated with a distributed transaction. When you issue a commit, changes to each database affected by a distributed transaction are made permanent. However, if a network or machine fails during the commit, the state of the distributed transaction might be unknown or in doubt. In that case, Oracle stores the text specified by COMMENT in the data dictionary along with the transaction ID. The text must be a quoted literal up to 50 characters long. An example follows:

```
COMMIT COMMENT 'In-doubt order transaction; notify Order Entry';
```

PL/SQL does not support the FORCE clause, which, in SQL, manually commits an in-doubt distributed transaction. For example, the following COMMIT statement is illegal:

```
COMMIT FORCE '23.51.54';  -- illegal
```

**Undoing Changes with ROLLBACK**

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. Rolling back is useful for two reasons. First, if you make a mistake like deleting the wrong row from a table, a rollback restores the original data. Second, if you start a transaction that you cannot finish because an exception is raised or a SQL statement fails, a rollback lets you return to the starting point to take corrective action and perhaps try again.

Consider the example below, in which you insert information about an employee into three different database tables. All three tables have a column that holds employee numbers and is constrained by a unique index. If an INSERT statement tries to store a duplicate employee number, the predefined exception DUP_VAL_ON_INDEX is raised. In that case, you want to undo all changes, so you issue a rollback in the exception handler.
DECLARE
    emp_id  INTEGER;
    ...
BEGIN
    SELECT empno, ... INTO emp_id, ... FROM new_emp WHERE ... 
    ...
    INSERT INTO emp VALUES (emp_id, ...);
    INSERT INTO tax VALUES (emp_id, ...);
    INSERT INTO pay VALUES (emp_id, ...);
    ...
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK;
    ...
END;

Statement-Level Rollbacks
Before executing a SQL statement, Oracle marks an implicit savepoint. Then, if the statement fails, Oracle rolls it back automatically. For example, if an INSERT statement raises an exception by trying to insert a duplicate value in a unique index, the statement is rolled back. Only work started by the failed SQL statement is lost. Work done before that statement in the current transaction is kept.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid schema objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

Undoing Partial Changes with SAVEPOINT
SAVEPOINT names and marks the current point in the processing of a transaction. Used with the ROLLBACK TO statement, savepoints let you undo parts of a transaction instead of the whole transaction. In the example below, you mark a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the empno column, the predefined exception DUP_VAL_ON_INDEX is raised. In that case, you roll back to the savepoint, undoing just the insert.

DECLARE
    emp_id  emp.empno%TYPE;
BEGIN

6-46 PL/SQL User’s Guide and Reference
UPDATE emp SET ... WHERE empno = emp_id;
DELETE FROM emp WHERE ...
...
SAVEPOINT do_insert;
INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO do_insert;
END;

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. However, the savepoint to which you roll back is not erased. For example, if you mark five savepoints, then roll back to the third, only the fourth and fifth are 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. However, you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers and can be reused within a transaction. This moves the savepoint from its old position to the current point in the transaction. Thus, a rollback to the savepoint affects only the current part of your transaction. An example follows:

BEGIN
  SAVEPOINT my_point;
  UPDATE emp SET ... WHERE empno = emp_id;
  ...
  SAVEPOINT my_point;  -- move my_point to current point
  INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK TO my_point;
END;

The number of active savepoints for each session is unlimited. An active savepoint is one marked since the last commit or rollback.

**How Oracle Does Implicit Rollbacks**

Before executing an INSERT, UPDATE, or DELETE statement, Oracle marks an implicit savepoint (unavailable to you). If the statement fails, Oracle rolls back to the savepoint. Normally, just the failed SQL statement is rolled back, not the whole
transformation. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to OUT parameters. Also, PL/SQL does not roll back database work done by the subprogram.

Ending Transactions

A good programming practice is to commit or roll back every transaction explicitly. Whether you issue the commit or rollback in your PL/SQL program or in the host environment depends on the flow of application logic. If you neglect to commit or roll back a transaction explicitly, the host environment determines its final state.

For example, in the SQL*Plus environment, if your PL/SQL block does not include a COMMIT or ROLLBACK statement, the final state of your transaction depends on what you do after running the block. If you execute a data definition, data control, or COMMIT statement or if you issue the EXIT, DISCONNECT, or QUIT command, Oracle commits the transaction. If you execute a ROLLBACK statement or abort the SQL*Plus session, Oracle rolls back the transaction.

In the Oracle Precompiler environment, if your program does not terminate normally, Oracle rolls back your transaction. A program terminates normally when it explicitly commits or rolls back work and disconnects from Oracle using the RELEASE parameter, as follows:

EXEC SQL COMMIT WORK RELEASE;

Setting Transaction Properties with SET TRANSACTION

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 against one or more tables 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 the example below, as a store manager, you use a read-only transaction to gather sales figures for the day, the past week, and the past month. The figures are unaffected by other users updating the database during the transaction.

DECLARE
    daily_sales   REAL;
Overview of Transaction Processing in PL/SQL

Interaction Between PL/SQL and Oracle

weekly_sales REAL;
monthly_sales REAL;
BEGIN
  ...
  COMMIT;  -- ends previous transaction
  SET TRANSACTION READ ONLY NAME 'Calculate sales figures';
  SELECT SUM(amt) INTO daily_sales FROM sales
    WHERE dte = SYSDATE;
  SELECT SUM(amt) INTO weekly_sales FROM sales
    WHERE dte > SYSDATE - 7;
  SELECT SUM(amt) INTO monthly_sales FROM sales
    WHERE dte > 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. Also, queries cannot be FOR UPDATE.

Overriding Default Locking

By default, Oracle locks data structures for you automatically. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table 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 before an update or delete is executed. However, Oracle automatically obtains row-level locks at update or delete time. So, use the FOR UPDATE clause only if you want to lock the rows before the update or delete.
Using FOR UPDATE

When you declare a cursor that will be referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks. An example follows:

```sql
DECLARE
    CURSOR c1 IS SELECT empno, sal FROM emp
                  WHERE job = 'SALESMAN' AND comm > sal
          FOR UPDATE NOWAIT;
```

The SELECT ... FOR UPDATE statement identifies the rows that will be updated or deleted, then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure the row is not changed by another user before the update.

The optional keyword NOWAIT tells Oracle not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the keyword NOWAIT, Oracle waits until the rows are available.

All rows are locked when you open the cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. So, you cannot fetch from a FOR UPDATE cursor after a commit. (For a workaround, see “Fetching Across Commits” on page 6-51.)

When querying multiple tables, you can use the FOR UPDATE clause to confine row locking to particular tables. Rows in a table are locked only if the FOR UPDATE OF clause refers to a column in that table. For example, the following query locks rows in the emp table but not in the dept table:

```sql
DECLARE
    CURSOR c1 IS SELECT ename, dname FROM emp, dept
                  WHERE emp.deptno = dept.deptno AND job = 'MANAGER'
          FOR UPDATE OF sal;
```

As the next example shows, you use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row fetched from a cursor:

```sql
DECLARE
    CURSOR c1 IS SELECT empno, job, sal FROM emp FOR UPDATE;
...;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO ...
    END LOOP;
```

...
... 
    UPDATE emp SET sal = new_sal WHERE CURRENT OF c1;
END LOOP;

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. For example, the statement below locks the emp table in row share mode. 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 emp IN ROW SHARE MODE NOWAIT;

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an exclusive lock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table. For more information about lock modes, see Oracle9i Application Developer’s Guide - Fundamentals.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row will one transaction wait for the other to complete.

Fetching Across Commits
The FOR UPDATE clause acquires exclusive row locks. All rows are locked when you open the cursor, and they are unlocked when you commit your transaction. So, you cannot fetch from a FOR UPDATE cursor after a commit. If you do, PL/SQL raises an exception. In the following example, the cursor FOR loop fails after the tenth insert:

DECLARE
    CURSOR c1 IS SELECT ename FROM emp FOR UPDATE OF sal;
    ctr NUMBER := 0;
BEGIN
    FOR emp_rec IN c1 LOOP  -- FETCHes implicitly
        ...
        ctr := ctr + 1;
        INSERT INTO temp VALUES (ctr, 'still going');
        IF ctr >= 10 THEN
            COMMIT;  -- releases locks
        END IF;
    END LOOP;
END;
If you want to fetch across commits, do not use the `FOR UPDATE` and `CURRENT OF` clauses. Instead, use the `ROWID` pseudocolumn to mimic the `CURRENT OF` clause. Simply select the rowid of each row into a `UROWID` variable. Then, use the rowid to identify the current row during subsequent updates and deletes. An example follows:

```sql
DECLARE
    CURSOR c1 IS SELECT ename, job, rowid FROM emp;
    my_ename emp.ename%TYPE;
    my_job emp.job%TYPE;
    my_rowid UROWID;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO my_ename, my_job, my_rowid;
        EXIT WHEN c1%NOTFOUND;
        UPDATE emp SET sal = sal * 1.05 WHERE rowid = my_rowid;
        -- this mimics WHERE CURRENT OF c1
        COMMIT;
    END LOOP;
    CLOSE c1;
END;
```

Be careful. In the last example, the fetched rows are not locked because no `FOR UPDATE` clause is used. So, other users might unintentionally overwrite your changes. Also, the cursor must have a read-consistent view of the data, so rollback segments used in the update are not released until the cursor is closed. This can slow down processing when many rows are updated.

The next example shows that you can use the `%ROWTYPE` attribute with cursors that reference the `ROWID` pseudocolumn:

```sql
DECLARE
    CURSOR c1 IS SELECT ename, sal, rowid FROM emp;
    emp_rec c1%ROWTYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO emp_rec;
        EXIT WHEN c1%NOTFOUND;
        ...
        IF ... THEN
            DELETE FROM emp WHERE rowid = emp_rec.rowid;
        END IF;
    END LOOP;
    CLOSE c1;
END;
```
Doing Independent Units of Work with Autonomous Transactions

A transaction is a series of SQL statements that does a logical unit of work. Often, one transaction starts another. In some applications, a transaction must operate outside the scope of the transaction that started it. This can happen, for example, when a transaction calls out to a data cartridge.

An autonomous transaction is an independent transaction started by another transaction, the main transaction. Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction. Figure 6–1 shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

Figure 6–1 Transaction Control Flow

Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. So, 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. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure’s autonomous operations, and the procedure need not know about the

```sql
PROCEDURE proc1 IS
  emp_id NUMBER;
BEGIN
  emp_id := 7788;
  INSERT ...
  SELECT ...
  proc2;
  DELETE ...
  COMMIT;
END;

PROCEDURE proc2 IS
  PRAGMA AUTON...
  dept_id NUMBER;
BEGIN
  dept_id := 20;
  UPDATE ...
  INSERT ...
  UPDATE ...
  COMMIT;
END;
```
Doing Independent Units of Work with Autonomous Transactions

application’s transaction context. That makes autonomous transactions less error-prone than regular transactions and easier to use.

Furthermore, autonomous transactions have all the functionality of regular transactions. They allow parallel queries, distributed processing, and all the transaction control statements including SET TRANSACTION.

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 functions and procedures
■ 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 follows:

PRAGMA AUTONOMOUS_TRANSACTION;

In the following example, you mark a packaged function as autonomous:

CREATE PACKAGE banking AS
    ...
    FUNCTION balance (acct_id INTEGER) RETURN REAL;
END banking;

CREATE PACKAGE BODY banking AS
    ...
    FUNCTION balance (acct_id INTEGER) RETURN REAL IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        my_bal REAL;
        BEGIN
            ...
        END;  
END banking;

Restriction: 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. For example, the following pragma is illegal:
CREATE PACKAGE banking AS
   PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
   ...  
   FUNCTION balance (acct_id INTEGER) RETURN REAL;
END banking;

In the next example, you mark a standalone procedure as autonomous:

CREATE PROCEDURE close_account (acct_id INTEGER, OUT balance) AS
   PRAGMA AUTONOMOUS_TRANSACTION;
   my_bal REAL;
BEGIN ... END;

In the following example, you mark a PL/SQL block as autonomous:

DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
   my_empno NUMBER(4);
BEGIN ... END;

Restriction: You cannot mark a nested PL/SQL block as autonomous.

In the example below, you mark a database trigger as autonomous. Unlike regular
triggers, autonomous triggers can contain transaction control statements such as
COMMIT and ROLLBACK.

CREATE TRIGGER parts_trigger
BEFORE INSERT ON parts FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO parts_log VALUES(:new.pnum, :new.pname);
   COMMIT; -- allowed only in autonomous triggers
END;

Autonomous Versus Nested Transactions

Although an autonomous transaction is started by another transaction, it is not a
nested transaction for the following reasons:

- 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**

As Figure 6–2 shows, the main transaction shares its context with nested routines, but not with autonomous transactions. Likewise, when one autonomous routine calls another (or itself recursively), the routines share no transaction context. However, when an autonomous routine calls a non-autonomous routine, the routines share the same transaction context.

**Figure 6–2  Transaction Context**

![Transaction Context Diagram](image)

**Transaction Visibility**

As Figure 6–3 on page 6-57 shows, changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. The changes also become visible to the main transaction when it resumes, but only if its isolation level is set to `READ COMMITTED` (the default).
If you set the isolation level of the main transaction to **SERIALIZABLE**, as follows, changes made by its autonomous transactions are *not* visible to the main transaction when it resumes:

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

![Figure 6–3 Transaction Visibility](image)

**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**
COMMIT ends the current transaction and makes permanent changes made during that transaction. ROLLBACK ends the current transaction and undoes changes made during that transaction. ROLLBACK TO undoes part of a transaction. SAVEPOINT names and marks the current point in a transaction. SET TRANSACTION sets transaction properties such as read/write access and isolation level.

**Note:** Transaction properties set in the main transaction apply only to that transaction, not to its autonomous transactions, and vice versa.

**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 called 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. As Figure 6–4 shows, when one transaction ends, the next SQL statement begins another transaction.

**Figure 6–4  Multiple Autonomous Transactions**

**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. So, when in an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does not roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

**Avoiding Errors**

To avoid some common errors, keep the following points in mind when designing autonomous transactions:

- 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. In that case, Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

- The Oracle initialization parameter `TRANSACTIONS` specifies the maximum number of concurrent transactions. That number might be exceeded if autonomous transactions (which run concurrently with the main transaction) are not taken into account.

- If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception. If the exception goes unhandled, the transaction is rolled back.

**Using Autonomous Triggers**

Among other things, you can use database triggers to log events transparently. Suppose you want to track all inserts into a table, even those that roll back. In the example below, you use a trigger to insert duplicate rows into a shadow table. Because it is autonomous, the trigger can commit inserts into the shadow table whether or not you commit inserts into the main table.

```sql
-- create a main table and its shadow table
CREATE TABLE parts (pnum NUMBER(4), pname VARCHAR2(15));
CREATE TABLE parts_log (pnum NUMBER(4), pname VARCHAR2(15));

-- create an autonomous trigger that inserts into the shadow table before each insert into the main table
```
CREATE TRIGGER parts_trig
BEFORE INSERT ON parts FOR EACH ROW
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO parts_log VALUES (:new.pnum, :new.pname);
    COMMIT;
END;

-- insert a row into the main table, and then commit the insert
INSERT INTO parts VALUES (1040, 'Head Gasket');
COMMIT;

-- insert another row, but then roll back the insert
INSERT INTO parts VALUES (2075, 'Oil Pan');
ROLLBACK;

-- show that only committed inserts add rows to the main table
SELECT * FROM parts ORDER BY pnum;

  PNUM  PNAME
  ------  ---------------
    1040   Head Gasket

-- show that both committed and rolled-back inserts add rows
-- to the shadow table
SELECT * FROM parts_log ORDER BY pnum;

  PNUM  PNAME
  ------  ---------------
    1040   Head Gasket
      2075   Oil Pan

Unlike regular triggers, autonomous triggers can execute DDL statements using
native dynamic SQL (discussed in Chapter 11). In the following example, trigger
bonus_trig drops a temporary database table after table bonus is updated:

CREATE TRIGGER bonus_trig
AFTER UPDATE ON bonus
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION; -- enables trigger to perform DDL
BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE temp_bonus';
END;

For more information about database triggers, see Oracle9i Application Developer's
Guide - Fundamentals.
Calling Autonomous Functions from SQL

A function called from SQL statements must obey certain rules meant to control side effects. (See "Controlling Side Effects of PL/SQL Subprograms" on page 8-9.) To check for violations of the rules, you can use the pragma RESTRICT_REFERENCES. The pragma asserts that a function does not read and/or write database tables and/or package variables. (For more information, See Oracle9i Application Developer’s Guide - Fundamentals.)

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 the example below shows. When you call 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."

```sql
-- create the debug table
CREATE TABLE debug_output (msg VARCHAR2(200));

-- create the package spec
CREATE PACKAGE debugging AS
    FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
    PRAGMA RESTRICT_REFERENCES(log_msg, WNDS, RNDS);
END debugging;

-- create the package body
CREATE PACKAGE BODY debugging AS
    FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        BEGIN
            -- the following insert does not violate the constraint
            -- WNDS because this is an autonomous routine
            INSERT INTO debug_output VALUES (msg);
            COMMIT;
        RETURN msg;
    END;
END debugging;

-- call the packaged function from a query
DECLARE
    my_empno NUMBER(4);
    my_ename VARCHAR2(15);
BEGIN
    ... select debugging.log_msg(ename) INTO my_ename FROM emp
        WHERE empno = my_empno;
```
Ensuring Backward Compatibility of PL/SQL Programs

PL/SQL Version 2 allows some abnormal behavior that are no longer allowed. Specifically, Version 2 lets you:

- Make forward references to RECORD and TABLE types when declaring variables
- Specify the name of a variable (not a datatype) in the RETURN clause of a function spec
- Assign values to the elements of an index-by table IN parameter
- Pass the fields of a record IN parameter to another subprogram as OUT parameters
- Use the fields of a record OUT parameter on the right-hand side of an assignment statement
- Use OUT parameters in the FROM list of a SELECT statement

For backward compatibility, you might want to keep this particular Version 2 behavior. You can do that by setting the PLSQL_V2_COMPATIBILITY flag. On the server side, you can set the flag in two ways:

- Add the following line to the Oracle initialization file:
  `PLSQL_V2_COMPATIBILITY=TRUE`
- Execute one of the following SQL statements:
  `ALTER SESSION SET PLSQL_V2_COMPATIBILITY = TRUE;`
  `ALTER SYSTEM SET PLSQL_V2_COMPATIBILITY = TRUE;`

If you specify FALSE (the default), only the current default behavior is allowed.

On the client side, a command-line option sets the flag. For example, with the Oracle Precompilers, you specify the run-time option `DBMS` on the command line.

-- even if you roll back in this scope, the insert
-- into 'debug_output' remains committed because
-- it is part of an autonomous transaction
IF ... THEN
    ROLLBACK;
END IF;
END;
There is nothing more exhilarating than to be shot at without result. —Winston Churchill

Run-time errors arise from design faults, coding mistakes, hardware failures, and many other sources. Although you cannot anticipate all possible errors, you can plan to handle certain kinds of errors meaningful to your PL/SQL program.

With many programming languages, unless you disable error checking, a run-time error such as stack overflow or division by zero stops normal processing and returns control to the operating system. With PL/SQL, a mechanism called exception handling lets you "bulletproof" your program so that it can continue operating in the presence of errors.

This chapter discusses the following topics:

- Overview of PL/SQL Error Handling
- 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
- Tips for Handling PL/SQL Errors
Overview of PL/SQL Error Handling

In PL/SQL, a warning or error condition is called an *exception*. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include *division by zero* and *out of memory*. Some common internal exceptions have predefined names, such as `ZERO_DIVIDE` and `STORAGE_ERROR`. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named `insufficient_funds` to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions must be given names.

When an error occurs, an exception is *raised*. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by `RAISE` statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called *exception handlers*. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

In the example below, you calculate and store a price-to-earnings ratio for a company with ticker symbol XYZ. If the company has zero earnings, the predefined exception `ZERO_DIVIDE` is raised. This stops normal execution of the block and transfers control to the exception handlers. The optional `OTHERS` handler catches all exceptions that the block does not name specifically.

```sql
DECLARE
    pe_ratio NUMBER(3,1);
BEGIN
    SELECT price / earnings INTO pe_ratio FROM stocks
    WHERE symbol = 'XYZ';  -- might cause division-by-zero error
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
    COMMIT;
EXCEPTION  -- exception handlers begin
    WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
        INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
        COMMIT;
    ...
    WHEN OTHERS THEN   -- handles all other errors
        ROLLBACK;
END;  -- exception handlers and block end here
```
Advantages of PL/SQL Exceptions

The last example illustrates exception handling, not the effective use of `INSERT` statements. For example, a better way to do the insert follows:

```
INSERT INTO stats (symbol, ratio) 
  SELECT symbol, DECODE(earnings, 0, NULL, price / earnings) 
  FROM stocks WHERE symbol = 'XYZ';
```

In this example, a subquery supplies values to the `INSERT` statement. If earnings are zero, the function `DECODE` returns a null. Otherwise, `DECODE` returns the price-to-earnings ratio.

**Advantages of PL/SQL Exceptions**

Using exceptions for error handling has several advantages. Without exception handling, every time you issue a command, you must check for execution errors:

```
BEGIN
  SELECT ...  -- check for 'no data found' error
  SELECT ...  -- check for 'no data found' error
  SELECT ...  -- check for 'no data found' error
END;
```

Error processing is not clearly separated from normal processing; nor is it robust. If you neglect to code a check, the error goes undetected and is likely to cause other, seemingly unrelated errors.

With exceptions, you can handle errors conveniently without the need to code multiple checks, as follows:

```
BEGIN
  SELECT ...  
  SELECT ...  
  SELECT ...  
  ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
END;
```

Exceptions improve readability by letting you isolate error-handling routines. The primary algorithm is not obscured by error recovery algorithms. Exceptions also improve reliability. You need not worry about checking for an error at every point it might occur. Just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.
Predefined PL/SQL Exceptions

An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

To handle other Oracle errors, you can use the `OTHERS` handler. The functions `SQLCODE` and `SQLERRM` are especially useful in the `OTHERS` handler because they return the Oracle error code and message text. Alternatively, you can use the `pragma EXCEPTION_INIT` to associate exception names with Oracle error codes.

PL/SQL declares predefined exceptions globally in package `STANDARD`, which defines the PL/SQL environment. So, you need not declare them yourself. You can write handlers for predefined exceptions using the names in the following list:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Oracle Error</th>
<th>SQLCODE Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS INTO NULL</td>
<td>ORA-06530</td>
<td>-6530</td>
</tr>
<tr>
<td>CASE NOT FOUND</td>
<td>ORA-06592</td>
<td>-6592</td>
</tr>
<tr>
<td>COLLECTION IS NULL</td>
<td>ORA-06531</td>
<td>-6531</td>
</tr>
<tr>
<td>CURSOR_ALREADY_OPEN</td>
<td>ORA-06511</td>
<td>-6511</td>
</tr>
<tr>
<td>DUP_VAL ON INDEX</td>
<td>ORA-00001</td>
<td>-1</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>ORA-01001</td>
<td>-1001</td>
</tr>
<tr>
<td>INVALID NUMBER</td>
<td>ORA-01722</td>
<td>-1722</td>
</tr>
<tr>
<td>LOGIN DENIED</td>
<td>ORA-01017</td>
<td>-1017</td>
</tr>
<tr>
<td>NO DATA FOUND</td>
<td>ORA-01403</td>
<td>+100</td>
</tr>
<tr>
<td>NOT_LOGGED_ON</td>
<td>ORA-01012</td>
<td>-1012</td>
</tr>
<tr>
<td>PROGRAM ERROR</td>
<td>ORA-06501</td>
<td>-6501</td>
</tr>
<tr>
<td>ROWTYPE MISMATCH</td>
<td>ORA-06504</td>
<td>-6504</td>
</tr>
<tr>
<td>SELF IS NULL</td>
<td>ORA-06525</td>
<td>-30625</td>
</tr>
<tr>
<td>STORAGE ERROR</td>
<td>ORA-06500</td>
<td>-6500</td>
</tr>
<tr>
<td>SUBSCRIPT BEYOND COUNT</td>
<td>ORA-06533</td>
<td>-6533</td>
</tr>
<tr>
<td>SUBSCRIPT OUTSIDE LIMIT</td>
<td>ORA-06532</td>
<td>-6532</td>
</tr>
<tr>
<td>SYS_INVALID ROWID</td>
<td>ORA-01410</td>
<td>-1410</td>
</tr>
<tr>
<td>TIMEOUT ON RESOURCE</td>
<td>ORA-00051</td>
<td>-51</td>
</tr>
<tr>
<td>TOO MANY ROWS</td>
<td>ORA-01422</td>
<td>-1422</td>
</tr>
<tr>
<td>VALUE ERROR</td>
<td>ORA-06502</td>
<td>-6502</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>ORA-01476</td>
<td>-1476</td>
</tr>
</tbody>
</table>
Brief descriptions of the predefined exceptions follow:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Raised when ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_INTO_NULL</td>
<td>Your program attempts to assign values to the attributes of an uninitialized (atomically null) object.</td>
</tr>
<tr>
<td>CASE_NOT_FOUND</td>
<td>None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.</td>
</tr>
<tr>
<td>COLLECTION_IS_NULL</td>
<td>Your program attempts to apply collection methods other than EXISTS to an uninitialized (atomically null) 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>Your program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers. So, your program cannot open that cursor inside the loop.</td>
</tr>
<tr>
<td>DUP_VAL_ON_INDEX</td>
<td>Your program attempts to store duplicate values in a database column that is constrained by a unique index.</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>Your program attempts an illegal cursor operation such as closing an unopened cursor.</td>
</tr>
<tr>
<td>INVALID_NUMBER</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>Your program attempts to log on to Oracle with an invalid username and/or password.</td>
</tr>
<tr>
<td>NO_DATA_FOUND</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. SQL aggregate functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls an aggregate function never raises NO_DATA_FOUND. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.</td>
</tr>
<tr>
<td>NOT_LOGGED_ON</td>
<td>Your program issues a database call without being connected to Oracle.</td>
</tr>
<tr>
<td>Exception</td>
<td>Raised when ...</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PROGRAM_ERROR</td>
<td>PL/SQL has an internal problem.</td>
</tr>
<tr>
<td>ROWTYPE_MISMATCH</td>
<td>The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, 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>Your program attempts to call a MEMBER method on a null instance. That is, the built-in parameter SELF (which is always the first parameter passed to a MEMBER method) is null.</td>
</tr>
<tr>
<td>STORAGE_ERROR</td>
<td>PL/SQL runs out of memory or memory has been corrupted.</td>
</tr>
<tr>
<td>SUBSCRIPT_BEYOND_COUNT</td>
<td>Your 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>Your 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>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>A time-out occurs while Oracle is waiting for a resource.</td>
</tr>
<tr>
<td>TOO_MANY_ROWS</td>
<td>A SELECT INTO statement returns more than one row.</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>Your program attempts to divide a number by zero.</td>
</tr>
</tbody>
</table>
Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements.

Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION. In the following example, you declare an exception named past_due:

```
DECLARE
    past_due EXCEPTION;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules for PL/SQL Exceptions

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. So, the sub-block cannot reference the global exception unless it was declared in a labeled block, in which case the following syntax is valid:

```
block_label.exception_name
```

The following example illustrates the scope rules:

```
DECLARE
    past_due EXCEPTION;
    acct_num NUMBER;
BEGIN
    DECLARE       -- sub-block begins
        past_due EXCEPTION;  -- this declaration prevails
        acct_num NUMBER;
```

Handling PL/SQL Errors  7-7
BEGIN
  ...
  IF ... THEN
    RAISE past_due; -- this is not handled
  END IF;
END;  ----------- sub-block ends
EXCEPTION
  WHEN past_due THEN  -- does not handle RAISEd exception
    ...
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. Therefore, the RAISE statement and the
WHEN clause refer to different exceptions. To have the enclosing block handle the
raised exception, you must remove its declaration from the sub-block or define an
OTHERS handler.

**Associating a PL/SQL Exception with a Number: Pragma EXCEPTION_INIT**

To handle unnamed internal exceptions, you must use the OTHERS handler or the
pragma EXCEPTION_INIT. A pragma is a compiler directive, which can be thought
of as a parenthetical remark to the compiler. Pragmas (also called pseudoinstructions)
are processed at compile time, not at run time. For example, in the language Ada,
the following pragma tells the compiler to optimize the use of storage space:

```
pragma OPTIMIZE(SPACE);
```

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an
exception name with an Oracle error number. That lets you refer to any internal
exception by name and to write a specific handler for it.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block,
subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, Oracle_error_number);
```

where exception_name is the name of a previously declared exception. The
pragma must appear somewhere after the exception declaration in the same
declarative section, as shown in the following example:

```
DECLARE
  deadlock_detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
```
BEGIN
...
EXCEPTION
  WHEN deadlock_detected THEN
    -- handle the error
END;

Defining Your Own PL/SQL Error Messages: Procedure RAISE_APPLICATION_ERROR

Package DBMS_STANDARD, which is supplied with Oracle, provides language facilities that help your application interact with Oracle. For example, the procedure raise_application_error lets you issue user-defined error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To call raise_application_error, use the 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. Package DBMS_STANDARD is an extension of package STANDARD, so you need not qualify references to its contents.

An application can call raise_application_error only from an executing stored subprogram (or method). When called, raise_application_error ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

In the following example, you call raise_application_error if an employee’s salary is missing:

CREATE PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) AS
  curr_sal NUMBER;
BEGIN
  SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
  IF curr_sal IS NULL THEN
    /* Issue user-defined error message. */
    raise_application_error(-20101, 'Salary is missing');
  ELSE
    UPDATE emp SET sal = curr_sal + amount WHERE empno = emp_id;
  END IF;
END raise_salary;

The calling 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 my_emp_id and my_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(:my_emp_id, :my_amount);
EXCEPTION
    WHEN null_salary THEN
        INSERT INTO emp_audit VALUES (:my_emp_id, ...);
    ...
END;
END-EXEC;

This technique allows the calling 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
How PL/SQL Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle error number using EXCEPTION_INIT. However, other user-defined exceptions must be raised explicitly by RAISE statements.

Raising Exceptions with the RAISE Statement

PL/SQL blocks and subprograms should raise an exception only when an error makes it undesirable or impossible to finish processing. You can place RAISE statements for a given exception anywhere within the scope of that exception. In the following example, you alert your PL/SQL block to a user-defined exception named out_of_stock:

```plsql
DECLARE
    out_of_stock   EXCEPTION;
    number_on_hand NUMBER(4);
BEGIN
    ...
    IF number_on_hand < 1 THEN
        RAISE out_of_stock;
    END IF;
EXCEPTION
    WHEN out_of_stock THEN
        -- handle the 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 the following example shows:

```plsql
DECLARE
    acct_type INTEGER;
BEGIN
    ...
    IF acct_type NOT IN (1, 2, 3) THEN
        RAISE INVALID_NUMBER;  -- raise predefined exception
    END IF;
EXCEPTION
    WHEN INVALID_NUMBER THEN
        ROLLBACK;
```
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. In the latter case, PL/SQL returns an unhandled exception error to the host environment.

However, exceptions cannot propagate across remote procedure calls (RPCs). Therefore, a PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see "Defining Your Own PL/SQL Error Messages: Procedure RAISE_APPLICATION_ERROR" on page 7-9.

Figure 7–1, Figure 7–2, and Figure 7–3 illustrate the basic propagation rules.

**Figure 7–1** Propagation Rules: Example 1

```sql
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;
```
How PL/SQL Exceptions Propagate

**Figure 7–2 Propagation Rules: Example 2**

```plaintext
BEGIN
  BEGIN
    IF X = 1 THEN
      RAISE A;
    ELSIF X = 2 THEN
      RAISE B;
    ELSE
      RAISE C;
    END IF;
    ...
  EXCEPTION
    WHEN A THEN ...
  END;
  ...
END;
```

Exception B is handled, then control passes to the host environment.

Exception B propagates to the first enclosing block with an appropriate handler.

**Figure 7–3 Propagation Rules: Example 3**

```plaintext
BEGIN
  BEGIN
    IF X = 1 THEN
      RAISE A;
    ELSIF X = 2 THEN
      RAISE B;
    ELSE
      RAISE C;
    END IF;
    ...
  EXCEPTION
    WHEN A THEN ...
  END;
  ...
END;
```

Exception C has no handler, so an unhandled exception is returned to the host environment.
An exception can propagate beyond its scope, that is, beyond the block in which it was declared. Consider the following example:

```
BEGIN
  ...
  DECLARE  ---------- sub-block begins
   past_due EXCEPTION;
  BEGIN
    ...
    IF ... THEN
      RAISE past_due;
    END IF;
  END;  ------------- sub-block ends
EXCEPTION
  ...
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

Because the block in which exception `past_due` was declared has no handler for it, the exception propagates to the enclosing block. But, according to the scope rules, enclosing blocks cannot reference exceptions declared in a sub-block. So, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the calling application gets the following error:

ORA-06510: PL/SQL: unhandled user-defined exception

**Reraising a PL/SQL Exception**

Sometimes, you want to *reraise* an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, simply place a `RAISE` statement in the local handler, as shown in the following example:

```
DECLARE
  out_of_balance  EXCEPTION;
BEGIN
  ...
  BEGIN  ---------- sub-block begins
    ...
    IF ... THEN
      RAISE out_of_balance;  -- raise the exception
    END IF;
  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 exception_name1 THEN -- handler
        sequence_of_statements1
    WHEN exception_name2 THEN -- another handler
        sequence_of_statements2
    ...  
    WHEN OTHERS THEN           -- optional handler
        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.
Handling Raised PL/SQL Exceptions

As the following example shows, use of the OTHERS handler guarantees that no exception will go unhandled:

```sql
EXCEPTION
    WHEN ... THEN
        -- handle the error
    WHEN ... THEN
        -- handle the error
    WHEN OTHERS THEN
        -- handle all other errors
END;
```

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:

```sql
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.

Handling Exceptions Raised in Declarations

Exceptions can be raised in declarations by faulty initialization expressions. For example, the following declaration raises an exception because the constant `credit_limit` cannot store numbers larger than 999:

```sql
DECLARE
    credit_limit CONSTANT NUMBER(3) := 5000;  -- raises an exception
BEGIN
    ...
```
EXCEPTION
  WHEN OTHERS THEN -- cannot catch the exception
        ...
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 Handlers**

Only one exception at a time can be active in the exception-handling part of a block or subprogram. So, an exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for the newly raised exception. From there on, the exception propagates normally. Consider the following example:

```sql
EXCEPTION
  WHEN INVALID_NUMBER THEN
      INSERT INTO ... -- might raise DUP_VAL_ON_INDEX
      WHEN DUP_VAL_ON_INDEX THEN ... -- cannot catch the exception
END;
```

**Branching to or from an Exception Handler**

A **GOTO** statement cannot branch into an exception handler. Also, a **GOTO** statement cannot branch from an exception handler into the current block. For example, the following **GOTO** statement is illegal:

```sql
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';
      <<my_label>>
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
      pe_ratio := 0;
      GOTO my_label; -- illegal branch into current block
END;
```

However, a **GOTO** statement can branch from an exception handler into an enclosing block.
Handling Raised PL/SQL Exceptions

Retrieving the Error Code and Error Message: SQLCODE and SQLERRM

In an exception handler, you can use the built-in functions SQLCODE and SQLERRM to find out which error occurred and to get the associated error message. For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is no data found, in which case SQLCODE returns +100. SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

For user-defined exceptions, SQLCODE returns +1 and SQLERRM returns the message

User-Defined Exception

unless you used the pragma EXCEPTION_INIT to associate the exception name with an Oracle error number, in which case SQLCODE returns that error number and SQLERRM returns the corresponding error message. The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message

ORA-0000: normal, successful completion

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass negative error numbers to SQLERRM. In the following example, you pass positive numbers and so get unwanted results:

DECLARE
  ...
  err_msg VARCHAR2(100);
BEGIN
  /* Get all Oracle error messages. */
  FOR err_num IN 1..9999 LOOP
    err_msg := SQLERRM(err_num); -- wrong; should be -err_num
    INSERT INTO errors VALUES (err_msg);
  END LOOP;
END;

Passing a positive number to SQLERRM always returns the message user-defined exception unless you pass +100, in which case SQLERRM returns the message no data found. Passing a zero to SQLERRM always returns the message normal, successful completion.
Handling Raised PL/SQL Exceptions

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
    err_num NUMBER;
    err_msg VARCHAR2(100);
BEGIN
    ...
    EXCEPTION
        ...
        WHEN OTHERS THEN
            err_num := SQLCODE;
            err_msg := SUBSTR(SQLERRM, 1, 100);
            INSERT INTO errors VALUES (err_num, err_msg);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to err_msg. The functions SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

**Note:** When using pragma RESTRICT_REFERENCES to assert the purity of a stored function, you cannot specify the constraints WNPS and RNPS if the function calls SQLCODE or SQLERRM.

Catching Unhandled Exceptions

Remember, if it cannot find a handler for a raised exception, PL/SQL returns an unhandled exception error to the host environment, which determines the outcome. For example, in the Oracle Precompilers environment, any database changes made by a failed SQL statement or PL/SQL block are rolled back.

Unhandled exceptions can also affect subprograms. If you exit a subprogram successfully, PL/SQL assigns values to OUT parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to OUT parameters (unless they are NOCOPY parameters). Also, if a stored subprogram fails with an unhandled exception, PL/SQL does not roll back database work done by the subprogram.

You can avoid unhandled exceptions by coding an OTHERS handler at the topmost level of every PL/SQL program.
Tips for Handling PL/SQL Errors

In this section, you learn three techniques that increase flexibility.

Continuing after an Exception Is Raised

An exception handler lets you recover from an otherwise fatal error before exiting a block. But when the handler completes, the block is terminated. You cannot return to the current block from an exception handler. In the following example, if the SELECT INTO statement raises ZERO_DIVIDE, you cannot resume with the INSERT statement:

```sql
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
  WHERE symbol = 'XYZ';
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    ...
END;
```

You can still handle an exception for a statement, then continue with the next statement. Simply 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 terminates, the enclosing block continues to execute at the point where the sub-block ends. Consider the following example:

```sql
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  BEGIN  ---------- sub-block begins
    SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
    WHERE symbol = 'XYZ';
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      pe_ratio := 0;
  END;  ---------- sub-block ends
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
```

```sql
```
In this example, if the SELECT INTO statement raises a ZERO_DIVIDE exception, the local handler catches it and sets \texttt{pe\_ratio} to zero. Execution of the handler is complete, so the sub-block terminates, and execution continues with the \texttt{INSERT} statement.

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 with the \texttt{\%BULK\_EXCEPTIONS} Attribute" on page 5-38.

### 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.

Consider the example below. When the exception handler completes, the sub-block terminates, control transfers to the \texttt{LOOP} statement in the enclosing block, the sub-block starts executing again, and the transaction is retried. You might want to use a \texttt{FOR} or \texttt{WHILE} loop to limit the number of tries.

```sql
DECLARE
    name   VARCHAR2(20);
    ans1   VARCHAR2(3);
    ans2   VARCHAR2(3);
    ans3   VARCHAR2(3);
    suffix NUMBER := 1;
BEGIN
    ...  
    LOOP  -- could be FOR i IN 1..10 LOOP to allow ten tries
        BEGIN  -- sub-block begins
            SAVEPOINT start_transaction;  -- mark a savepoint
```
/* Remove rows from a table of survey results. */
DELETE FROM results WHERE answer1 = 'NO';
/* Add a survey respondent’s name and answers. */
INSERT INTO results VALUES (name, ans1, ans2, ans3);
-- 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

Exceptions can mask the statement that caused an error, as the following example shows:

BEGIN
    SELECT ... 
    SELECT ... 
    SELECT ... 
    ... 
EXCEPTION
    WHEN NO_DATA_FOUND THEN ...
        -- Which SELECT statement caused the error?
END;

Normally, this is not a problem. But, if the need arises, you can use a locator variable
to track statement execution, as follows:

DECLARE
    stmt INTEGER := 1;  -- designates 1st SELECT statement
BEGIN
    SELECT ... 
    stmt := 2;  -- designates 2nd SELECT statement
    SELECT ... 
    stmt := 3;  -- designates 3rd SELECT statement
    SELECT ... 
    ... 
EXCEPTION
WHEN NO_DATA_FOUND THEN
    INSERT INTO errors VALUES ('Error in statement ' || stmt);
END;
Civilization advances by extending the number of important operations that we can perform without thinking about them. —Alfred North Whitehead

This chapter shows you how to use subprograms, which let you name and encapsulate a sequence of statements. Subprograms aid application development by isolating operations. They are like building blocks, which you can use to construct modular, maintainable applications.

This chapter discusses the following topics:

- What Are Subprograms?
- Advantages of Subprograms
- Understanding PL/SQL Procedures
- Understanding PL/SQL Functions
- Declaring PL/SQL Subprograms
- Packaging PL/SQL Subprograms Together
- Actual Versus Formal Subprogram Parameters
- Positional Versus Named Notation for Subprogram Parameters
- Specifying Subprogram Parameter Modes
- Passing Large Data Structures with the NOCOPY Compiler Hint
- Using Default Values for Subprogram Parameters
- Understanding Subprogram Parameter Aliasing
- Overloading Subprogram Names
- How Subprogram Calls Are Resolved
- Accepting and Returning Multiple Rows with Table Functions
- Invoker Rights Versus Definer Rights
- Understanding and Using Recursion
- Calling External Subprograms
What Are Subprograms?

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called procedures and functions. Generally, you use a procedure to perform an action and a function to compute a value.

Like unnamed or anonymous PL/SQL blocks, subprograms have a declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local and cease to exist when you exit the subprogram. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution.

Consider the following procedure named debit_account, which debits a bank account:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
  old_balance REAL;
  new_balance REAL;
  overdrawn EXCEPTION;
BEGIN
  SELECT bal INTO old_balance FROM accts
    WHERE acct_no = acct_id;
  new_balance := old_balance - amount;
  IF new_balance < 0 THEN
    RAISE overdrawn;
  ELSE
    UPDATE accts SET bal = new_balance
      WHERE acct_no = acct_id;
  END IF;
EXCEPTION
  WHEN overdrawn THEN
    ...
END debit_account;
```

When invoked or called, this procedure accepts an account number and a debit amount. It uses the account number to select the account balance from the accts database table. Then, it uses the debit amount to compute a new balance. If the new balance is less than zero, an exception is raised; otherwise, the bank account is updated.
Advantages of Subprograms

Subprograms provide extensibility; that is, they let you tailor the PL/SQL language to suit your needs. For example, if you need a procedure that creates new departments, you can easily write one, as follows:

```sql
PROCEDURE create_dept (new_dname VARCHAR2, new_loc VARCHAR2) IS
    BEGIN
        INSERT INTO dept VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
    END create_dept;
```

Subprograms also provide modularity; that is, they let you break a program down into manageable, well-defined modules. This supports top-down design and the stepwise refinement approach to problem solving.

In addition, subprograms promote reusability and maintainability. Once validated, a subprogram can be used with confidence in any number of applications. If its definition changes, only the subprogram is affected. This simplifies maintenance.

Finally, subprograms aid abstraction, the mental process of deriving a universal from particulars. To use subprograms, you must know what they do, not how they work. Therefore, you can design applications from the top down without worrying about implementation details. Dummy subprograms (stubs) allow you to defer the definition of procedures and functions until you test and debug the main program.

Understanding PL/SQL Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the syntax:

```
[CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
    [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [local declarations]
    BEGIN
        executable statements
    [EXCEPTION
        exception handlers]
    END [name];
```

where parameter stands for the following syntax:

```
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype
    [{:= | DEFAULT} expression]
```
The CREATE clause lets you create standalone procedures, which are stored in an Oracle database. You can execute the CREATE PROCEDURE statement interactively from SQL*Plus or from a program using native dynamic SQL (see Chapter 11).

The AUTHID clause determines whether a stored procedure executes with the privileges of its owner (the default) or current user and whether its unqualified references to schema objects are resolved in the schema of the owner or current user. You can override the default behavior by specifying CURRENT_USER. For more information, see "Invoker Rights Versus Definer Rights" on page 8-47.

The pragma AUTONOMOUS_TRANSACTION instructs the PL/SQL compiler to mark a procedure as autonomous (independent). Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction. For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-53.

You cannot constrain the datatype of a parameter. For example, the following declaration of acct_id is illegal because the datatype CHAR is size-constrained:

```
PROCEDURE reconcile (acct_id CHAR(5)) IS ... -- illegal
```

However, you can use the following workaround to size-constrain parameter types indirectly:

```
DECLARE
    SUBTYPE Char5 IS CHAR(5);
    PROCEDURE reconcile (acct_id Char5) IS ...
```

A procedure has two parts: the specification (spec for short) and the body. The procedure spec begins with the keyword PROCEDURE and ends with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses.

The procedure body begins with the keyword IS (or AS) and ends with the keyword END followed by an optional procedure name. The procedure body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE, which introduces declarations in an anonymous PL/SQL block, is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). At least one statement must appear in the executable part of a procedure. The NULL
Understanding PL/SQL Procedures

statement meets this requirement. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Consider the procedure raise_salary, which increases the salary of an employee by a given amount:

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
  current_salary REAL;
  salary_missing EXCEPTION;
BEGIN
  SELECT sal INTO current_salary FROM emp
  WHERE empno = emp_id;
  IF current_salary IS NULL THEN
    RAISE salary_missing;
  ELSE
    UPDATE emp SET sal = sal + amount
    WHERE empno = emp_id;
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    INSERT INTO emp_audit VALUES (emp_id, 'No such number');
  WHEN salary_missing THEN
    INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

When called, this procedure accepts an employee number and a salary increase amount. It uses the employee number to select the current salary from the emp database table. If the employee number is not found or if the current salary is null, an exception is raised. Otherwise, the salary is updated.

A procedure is called as a PL/SQL statement. For example, you might call the procedure raise_salary as follows:

```
raise_salary(emp_id, amount);
```
Understanding PL/SQL Functions

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause. You write (local) functions using the syntax:

```
[CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[, parameter]...)] RETURN datatype
[AUTHID {DEFINER | CURRENT_USER}]
[PARALLEL_ENABLE
  [{{CLUSTER parameter BY (column_name [, column_name]...)}|]
  [ORDER parameter BY (column_name [, column_name]...)]
  [{{PARTITION parameter BY([ (RANGE|HASH) (column_name [, column_name]...)]|ANY))]
  ]
  [DETERMINISTIC] {IS | AS}
  [PIPELINED [USING implementation_type]]
  [AGGREGATE [UPDATE VALUE] {WITH EXTERNAL CONTEXT} USING implementation_type]
  [PRAGMA AUTONOMOUS_TRANSACTION;]
[local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
```

The CREATE clause lets you create standalone functions, which are stored in an Oracle database. You can execute the CREATE FUNCTION statement interactively from SQL*Plus or from a program using native dynamic SQL.

The AUTHID clause determines whether a stored function executes with the privileges of its owner (the default) or current user and whether its unqualified references to schema objects are resolved in the schema of the owner or current user. You can override the default behavior by specifying CURRENT_USER.

The PARALLEL_ENABLE option declares that a stored function can be used safely in the slave sessions of parallel DML evaluations. The state of a main (logon) session is never shared with slave sessions. Each slave session has its own state, which is
initialized when the session begins. The function result should not depend on the state of session (static) variables. Otherwise, results might vary across sessions.

The hint `DETERMINISTIC` helps the optimizer avoid redundant function calls. If a stored function was called previously with the same arguments, the optimizer can elect to use the previous result. The function result should not depend on the state of session variables or schema objects. Otherwise, results might vary across calls. Only `DETERMINISTIC` functions can be called from a function-based index or a materialized view that has query-rewrite enabled. For more information, see Oracle9i SQL Reference.

The pragma `AUTONOMOUS_TRANSACTION` instructs the PL/SQL compiler to mark a function as autonomous (independent). Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.

You cannot constrain (with `NOT NULL` for example) the datatype of a parameter or a function return value. However, you can use a workaround to size-constrain them indirectly. See "Understanding PL/SQL Procedures" on page 8-3.

Like a procedure, a function has two parts: the spec and the body. The function spec begins with the keyword `FUNCTION` and ends with the `RETURN` clause, which specifies the datatype of the return value. Parameter declarations are optional. Functions that take no parameters are written without parentheses.

The function body begins with the keyword `IS` (or `AS`) and ends with the keyword `END` followed by an optional function name. The function body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords `IS` and `BEGIN`. The keyword `DECLARE` is not used. The executable part contains statements, which are placed between the keywords `BEGIN` and `EXCEPTION` (or `END`). One or more `RETURN` statements must appear in the executable part of a function. The exception-handling part contains exception handlers, which are placed between the keywords `EXCEPTION` and `END`.

Consider the function `sal_ok`, which determines if a salary is out of range:

```
FUNCTION sal_ok (salary REAL, title VARCHAR2) RETURN BOOLEAN IS
  min_sal REAL;
  max_sal REAL;
BEGIN
  SELECT losal, hisal INTO min_sal, max_sal FROM sals
  WHERE job = title;
  RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;
```
When called, this function accepts an employee salary and job title. It uses the job title to select range limits from the sals database table. The function identifier, sal_ok, is set to a Boolean value by the RETURN statement. If the salary is out of range, sal_ok is set to FALSE; otherwise, sal_ok is set to TRUE.

A function is called as part of an expression, as the example below shows. The function identifier sal_ok acts like a variable whose value depends on the parameters passed to it.

IF sal_ok(new_sal, new_title) THEN ... 

Using the RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. (Do not confuse the RETURN statement with the RETURN clause in a function spec, which specifies the datatype of the return value.)

A subprogram can contain several RETURN statements. The last lexical statement does not need to be a RETURN statement. Executing any RETURN statement completes the subprogram immediately. However, to have multiple exit points in a subprogram is a poor programming practice.

In procedures, a RETURN statement cannot return a value, and therefore cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a RETURN statement must contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable of the type specified in the RETURN clause. Observe how the function balance returns the balance of a specified bank account:

FUNCTION balance (acct_id INTEGER) RETURN REAL IS
  acct_bal REAL;
BEGIN
  SELECT bal INTO acct_bal FROM accts
  WHERE acct_no = acct_id;
  RETURN acct_bal;
END balance;

The following example shows that the expression in a function RETURN statement can be arbitrarily complex:

FUNCTION compound ( 
years NUMBER,
amount NUMBER,
rate NUMBER) RETURN NUMBER IS
BEGIN
    RETURN amount * POWER((rate / 100) + 1, years);
END compound;

In a function, there must be at least one execution path that leads to a RETURN statement. Otherwise, you get a function returned without value error at run time.

Controlling Side Effects of PL/SQL Subprograms

To be callable from SQL statements, a stored function must obey the following "purity" rules, which are meant to control side effects:

- When called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
- When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.
- When called 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 violations of the rules, you can use the pragma (compiler directive) RESTRICT_REFERENCES. The pragma asserts that a function does not read and/or write database tables and/or package variables. For example, the following pragma asserts that packaged function credit_ok writes no database state (WNDS) and reads no package state (RNPS):

CREATE PACKAGE loans AS
...
    FUNCTION credit_ok RETURN BOOLEAN;
    PRAGMA RESTRICT_REFERENCES (credit_ok, WNDS, RNPS);
END loans;

Note: A static INSERT, UPDATE, or DELETE statement always violates WNDS. It also violates RNDS (reads no database state) if it reads any columns. A dynamic INSERT, UPDATE, or DELETE statement always violates WNDS and RNDS.
Declaring PL/SQL Subprograms

You can declare subprograms in any PL/SQL block, subprogram, or package. But, you must declare subprograms at the end of a declarative section after all other program items.

PL/SQL requires that you declare an identifier before using it. Therefore, you must declare a subprogram before calling it. For example, the following declaration of procedure `award_bonus` is illegal because `award_bonus` calls procedure `calc_rating`, which is not yet declared when the call is made:

```plsql
DECLARE
  ... PROCEDURE award_bonus IS
    BEGIN
      calc_rating(...);  -- undeclared identifier
    ... END;

  PROCEDURE calc_rating (...) IS
    BEGIN
      ... END;

In this case, you can solve the problem easily by placing procedure `calc_rating` before procedure `award_bonus`. However, the easy solution does not always work. For example, suppose the procedures are mutually recursive (call each other) or you want to define them in logical or alphabetical order.

You can solve the problem by using a special subprogram declaration called a forward declaration, which consists of a subprogram spec terminated by a semicolon. In the following example, the forward declaration advises PL/SQL that the body of procedure `calc_rating` can be found later in the block.

```plsql
DECLARE
  PROCEDURE calc_rating ( ... );  -- forward declaration

Although the formal parameter list appears in the forward declaration, it must also appear in the subprogram body. You can place the subprogram body anywhere after the forward declaration, but they must appear in the same program unit.
Packaging PL/SQL Subprograms Together

You can group logically related subprograms in a package, which is stored in the database. That way, the subprograms can be shared by many applications. The subprogram specs go in the package spec, and the subprogram bodies go in the package body, where they are invisible to applications. Thus, packages allow you to hide implementation details. An example follows:

```sql
CREATE PACKAGE emp_actions AS  -- package spec
    PROCEDURE hire_employee (emp_id INTEGER, name VARCHAR2, ...);
    PROCEDURE fire_employee (emp_id INTEGER);
    PROCEDURE raise_salary (emp_id INTEGER, amount REAL);
    ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS  -- package body
    PROCEDURE hire_employee (emp_id INTEGER, name VARCHAR2, ...) IS
        BEGIN
            INSERT INTO emp VALUES (emp_id, name, ...);
            END hire_employee;
        PROCEDURE fire_employee (emp_id INTEGER) IS
            BEGIN
                DELETE FROM emp WHERE empno = emp_id;
                END fire_employee;
        PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
            BEGIN
                UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
                END raise_salary;
            ...
END emp_actions;
```

You can define subprograms in a package body without declaring their specs in the package spec. However, such subprograms can be called only from inside the package. For more information about packages, see Chapter 9.
Subprograms pass information using parameters. The variables or expressions referenced in the parameter list of a subprogram call are actual parameters. For example, the following procedure call lists two actual parameters named emp_num and amount:

```sql
raise_salary(emp_num, amount);
```

The next procedure call shows that expressions can be used as actual parameters:

```sql
raise_salary(emp_num, merit + cola);
```

The variables declared in a subprogram spec and referenced in the subprogram body are formal parameters. For example, the following procedure declares two formal parameters named emp_id and amount:

```sql
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
BEGIN
  UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
END raise_salary;
```

A good programming practice is to use different names for actual and formal parameters.

When you call procedure raise_salary, the actual parameters are evaluated and the results are assigned to the corresponding formal parameters. If necessary, before assigning the value of an actual parameter to a formal parameter, PL/SQL converts the datatype of the value. For example, the following call to raise_salary is valid:

```sql
raise_salary(emp_num, '2500');
```

The actual parameter and its corresponding formal parameter must have compatible datatypes. For instance, PL/SQL cannot convert between the DATE and REAL datatypes. Also, the result must be convertible to the new datatype. The following procedure call raises the predefined exception VALUE_ERROR because PL/SQL cannot convert the second actual parameter to a number:

```sql
raise_salary(emp_num, '$2500'); -- note the dollar sign
```
Positional Versus Named Notation for Subprogram Parameters

When calling a subprogram, you can write the actual parameters using either positional or named notation. That is, you can indicate the association between an actual and formal parameter by position or name. So, given the declarations

```
DECLARE
    acct INTEGER;
    amt  REAL;
    PROCEDURE credit_acct (acct_no INTEGER, amount REAL) IS ... 
```

you can call the procedure `credit_acct` in four logically equivalent ways:

```
BEGIN
    credit_acct(acct, amt);          -- positional notation
    credit_acct(amount => amt, acct_no => acct); -- named notation
    credit_acct(acct_no => acct, amount => amt); -- named notation
    credit_acct(acct, amount => amt);     -- mixed notation
```

Using Positional Notation

The first procedure call uses positional notation. The PL/SQL compiler associates the first actual parameter, `acct`, with the first formal parameter, `acct_no`. And, the compiler associates the second actual parameter, `amt`, with the second formal parameter, `amount`.

Using Named Notation

The second procedure call uses named notation. An arrow (`=>`) serves as the association operator, which associates the formal parameter to the left of the arrow with the actual parameter to the right of the arrow.

The third procedure call also uses named notation and shows that you can list the parameter pairs in any order. So, you need not know the order in which the formal parameters are listed.

Using Mixed Notation

The fourth procedure call shows that you can mix positional and named notation. In this case, the first parameter uses positional notation, and the second parameter uses named notation. Positional notation must precede named notation. The reverse is not allowed. For example, the following procedure call is illegal:

```
credit_acct(acct_no => acct, amt); -- illegal
```
Specifying Subprogram Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, \textit{IN} (the default), \textit{OUT}, and \textit{IN OUT}, can be used with any subprogram. However, avoid using the \textit{OUT} and \textit{IN OUT} modes with functions. The purpose of a function is to take zero or more arguments (actual parameters) and return a single value. To have a function return multiple values is a poor programming practice. Also, functions should be free from \textit{side effects}, which change the values of variables not local to the subprogram.

Using the \textit{IN} Mode

An \textit{IN} parameter lets you pass values to the subprogram being called. Inside the subprogram, an \textit{IN} parameter acts like a constant. Therefore, it cannot be assigned a value. For example, the following assignment statement causes a compilation error:

```
PROCEDURE debit_account (acct_id IN INTEGER, amount IN REAL) IS
  minimum_purchase CONSTANT REAL DEFAULT 10.0;
  service_charge CONSTANT REAL DEFAULT 0.50;
BEGIN
  IF amount < minimum_purchase THEN
    amount := amount + service_charge; -- causes compilation error
  END IF;
  ...
END debit_account;
```

The actual parameter that corresponds to an \textit{IN} formal parameter can be a constant, literal, initialized variable, or expression. Unlike \textit{OUT} and \textit{IN OUT} parameters, \textit{IN} parameters can be initialized to default values. For more information, see “Using Default Values for Subprogram Parameters” on page 8-19.

Using the \textit{OUT} Mode

An \textit{OUT} parameter lets you return values to the caller of a subprogram. Inside the subprogram, an \textit{OUT} parameter acts like a variable. That means you can use an \textit{OUT} formal parameter as if it were a local variable. You can change its value or reference the value in any way, as the following example shows:

```
PROCEDURE calc_bonus (emp_id IN INTEGER, bonus OUT REAL) IS
  hire_date DATE;
  bonus_missing EXCEPTION;
BEGIN
  SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
  WHERE empno = emp_id;
```

```
```
IF bonus IS NULL THEN
    RAISE bonus_missing;
END IF;
IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
    bonus := bonus + 500;
END IF;
...
EXCEPTION
    WHEN bonus_missing THEN
      ...
END calc_bonus;

The actual parameter that corresponds to an OUT formal parameter must be a variable; it cannot be a constant or an expression. For example, the following procedure call is illegal:

calc_bonus(7499, salary + commission);  -- causes compilation error

An OUT actual parameter can have a value before the subprogram is called. However, when you call the subprogram, the value is lost unless you specify the compiler hint NOCOPY (see “Passing Large Data Structures with the NOCOPY Compiler Hint” on page 8-17) or the subprogram exits with an unhandled exception.

Like variables, OUT formal parameters are initialized to NULL. So, the datatype of an OUT formal parameter cannot be a subtype defined as NOT NULL (that includes the built-in subtypes NATURALN and POSITIVEN). Otherwise, when you call the subprogram, PL/SQL raises VALUE_ERROR. An example follows:

DECLARE
    SUBTYPE Counter IS INTEGER NOT NULL;
    rows Counter := 0;
PROCEDURE count_emps (n OUT Counter) IS
  BEGIN
    SELECT COUNT(*) INTO n FROM emp;
  END;
BEGIN
  count_emps(rows);  -- raises VALUE_ERROR

Before exiting a subprogram, explicitly 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. However, if you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.
Using the IN OUT Mode

An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, an IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable.

The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or an expression.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. However, 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 all you need to know about the parameter modes.

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
<th>IN OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>the default</td>
<td>must be specified</td>
<td>must be specified</td>
</tr>
<tr>
<td>passes values to a subprogram</td>
<td>returns values to the caller</td>
<td>passes initial values to a subprogram and returns updated values to the caller</td>
</tr>
<tr>
<td>formal parameter acts like a constant</td>
<td>formal parameter acts like a variable</td>
<td>formal parameter acts like an initialized 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>
</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>
</tr>
<tr>
<td>actual parameter is passed by reference (a pointer to the value is passed in)</td>
<td>actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified</td>
<td>actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified</td>
</tr>
</tbody>
</table>
Suppose a subprogram declares an **IN** parameter, an **OUT** parameter, and an **IN OUT** parameter. When you call the subprogram, the **IN** parameter is passed by reference. That is, a pointer to the **IN** actual parameter is passed to the corresponding formal parameter. So, both parameters reference the same memory location, which holds the value of the actual parameter.

By default, the **OUT** and **IN OUT** parameters are passed by value. That is, the value of the **IN OUT** actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the **OUT** and **IN OUT** formal parameters are copied into the corresponding actual parameters.

When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory. To prevent that, you can specify the **NOCOPY** hint, which allows the PL/SQL compiler to pass **OUT** and **IN OUT** parameters by reference.

In the following example, you ask the compiler to pass **IN OUT** parameter `my_staff` by reference instead of by value:

```sql
DECLARE
  TYPE Staff IS VARRAY(200) OF Employee;
  PROCEDURE reorganize (my_staff IN OUT NOCOPY Staff) IS ...

Remember, **NOCOPY** is a hint, not a directive. So, the compiler might pass `my_staff` by value despite your request. Usually, however, **NOCOPY** succeeds. So, it can benefit any PL/SQL application that passes around large data structures.

In the example below, 25000 records are loaded into a local nested table, which is passed to two local procedures that do nothing but execute `NULL` statements. However, a call to one procedure takes 21 seconds because of all the copying. With **NOCOPY**, a call to the other procedure takes much less than 1 second.

```
PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
BEGIN NULL; END;
BEGIN
SELECT * INTO emp_tab(1) FROM emp WHERE empno = 7788;
emp_tab.EXTEND(24999, 1); -- copy element 1 into 2..25000
get_time(t1);
do_nothing1(emp_tab); -- pass IN OUT parameter
get_time(t2);
do_nothing2(emp_tab); -- pass IN OUT NOCOPY parameter
get_time(t3);
dbms_output.put_line('Call Duration (secs)');
dbms_output.put_line('--------------------');
dbms_output.put_line('Just IN OUT: ' || TO_CHAR(t2 - t1));
dbms_output.put_line('With NOCOPY: ' || TO_CHAR(t3 - t2));
END;
SQL> /
Call Duration (secs)
--------------------
Just IN OUT: 21
With NOCOPY: 0

The Trade-Off for Better Performance with NOCOPY

NOCOPY lets you trade well-defined exception semantics for better performance. Its use affects exception handling in the following ways:

- Because NOCOPY is a hint, not a directive, the compiler can pass NOCOPY parameters to a subprogram by value or by reference. So, if the subprogram exits with an unhandled exception, you cannot rely on the values of the NOCOPY actual parameters.

- By default, if a subprogram exits with an unhandled exception, the values assigned to its OUT and IN OUT formal parameters are not copied into the corresponding actual parameters, and changes appear to roll back. However, when you specify NOCOPY, assignments to the formal parameters immediately affect the actual parameters as well. So, if the subprogram exits with an unhandled exception, the (possibly unfinished) changes are not "rolled back."

- Currently, RPC protocol lets you pass parameters only by value. So, exception semantics can change silently when you partition applications. For example, if you move a local procedure with NOCOPY parameters to a remote site, those parameters will no longer be passed by reference.

Also, the use of NOCOPY increases the likelihood of parameter aliasing. For more information, see "Understanding Subprogram Parameter Aliasing" on page 8-21.
Restrictions on NOCOPY

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 index-by table. This restriction does not apply to entire index-by tables.
- The actual parameter is constrained (by scale or NOT NULL for example). This restriction does not extend to constrained elements or attributes. Also, it does not apply to size-constrained character strings.
- The actual and formal parameters are records, one or both records were declared using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit datatype conversion.
- The subprogram is involved in an external or remote procedure call.

Using Default Values for Subprogram Parameters

As the example below shows, you can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

```plsql
PROCEDURE create_dept (    
    new_dname VARCHAR2 DEFAULT 'TEMP',
    new_loc   VARCHAR2 DEFAULT 'TEMP') IS
BEGIN
    INSERT INTO dept
    VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
    ... 
END;
```

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the following calls to `create_dept`:

```plsql
create_dept;
create_dept('MARKETING');
create_dept('MARKETING', 'NEW YORK');
```
The first call passes no actual parameters, so both default values are used. The second call passes one actual parameter, so the default value for new_loc is used. The third call passes two actual parameters, so neither default value is used.

Usually, you can use positional notation to override the default values of formal parameters. However, you cannot skip a formal parameter by leaving out its actual parameter. For example, the following call incorrectly associates the actual parameter 'NEW YORK' with the formal parameter new_dname:

```plsql
create_dept('NEW YORK');  -- incorrect
```

You cannot solve the problem by leaving a placeholder for the actual parameter. For example, the following call is not allowed:

```plsql
create_dept(, 'NEW YORK');  -- not allowed
```

In such cases, you must use named notation, as follows:

```plsql
create_dept(new_loc => 'NEW YORK');
```

Also, you cannot assign a null to an uninitialized formal parameter by leaving out its actual parameter. For example, given the declaration

```plsql
DECLARE
  FUNCTION gross_pay (
    emp_id   IN NUMBER,
    st_hours IN NUMBER DEFAULT 40,
    ot_hours IN NUMBER) RETURN REAL IS
  BEGIN
    ...
  END;
```

the following function call does not assign a null to ot_hours:

```plsql
IF gross_pay(emp_num) > max_pay THEN ...  -- not allowed
```

Instead, you must pass the null explicitly, as in

```plsql
IF gross_pay(emp_num, ot_hour => NULL) > max_pay THEN ...
```

or you can initialize ot_hours to NULL, as follows:

```plsql
ot_hours IN NUMBER DEFAULT NULL;
```

Finally, when creating a stored subprogram, you cannot use host variables (bind variables) in the DEFAULT clause. The following SQL*Plus example causes a bad
bind variable error because at the time of creation, num is just a placeholder whose value might change:

```
SQL> VARIABLE num NUMBER
SQL> CREATE FUNCTION gross_pay (emp_id IN NUMBER DEFAULT :num, ...
```

### Understanding 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, in which case 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 the example below, procedure add_entry refers to varray lexicon in two different ways: as a parameter and as a global variable. So, when add_entry is called, the identifiers word_list and lexicon name the same varray.

```
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);
    -- prints 'aardvark' if parameter was passed by value
    -- prints 'aardwolf' if parameter was passed by reference
END;
```
The result depends on the method of parameter passing chosen by the compiler. If the compiler chooses the by-value method, `word_list` and `lexicon` are separate copies of the same varray. So, changing one does not affect the other. But, if the compiler chooses the by-reference method, `word_list` and `lexicon` are just different names for the same varray. (Hence, the term "aliasing.") So, changing the value of `lexicon(1)` also changes the value of `word_list(1)`.

Aliasing can also occur when the same actual parameter appears more than once in a subprogram call. In the example below, `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.

```plsql
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. Consider the example below. After the assignment, `emp_cv2` is an alias of `emp_cv1` because both point to the same query work area. So, both can alter its state. That is why the first fetch from `emp_cv2` fetches the third row (not the first) and why the second fetch from `emp_cv2` fails after you close `emp_cv1`.

```plsql
PROCEDURE get_emp_data ( emp_cv1 IN OUT EmpCurTyp, emp_cv2 IN OUT EmpCurTyp) IS
    emp_rec emp%ROWTYPE;
BEGIN
    OPEN emp_cv1 FOR SELECT * FROM emp;
    emp_cv2 := emp_cv1;
    FETCH emp_cv1 INTO emp_rec;  -- fetches first row
    FETCH emp_cv2 INTO emp_rec;  -- fails
END;
```
Overloading Subprogram Names

PL/SQL lets you *overload* subprogram names and type methods. That is, you can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

Suppose you want to initialize the first $n$ rows in two index-by tables that were declared as follows:

```sql
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
    hiredate_tab DateTabTyp;
    sal_tab RealTabTyp;
BEGIN
    ...
END;
```

You might write the following procedure to initialize the index-by table named `hiredate_tab`:

```sql
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := SYSDATE;
    END LOOP;
END initialize;
```

And, you might write the next procedure to initialize the index-by table named `sal_tab`:

```sql
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := 0.0;
    END LOOP;
END initialize;
```
Because the processing in these two procedures is the same, it is logical to give them the same name.

You can place the two overloaded initialize procedures in the same block, subprogram, or package. PL/SQL determines which of the two procedures is being called by checking their formal parameters. In the following example, the version of initialize that PL/SQL uses depends on whether you call the procedure with a DateTabTyp or RealTabTyp parameter:

```plsql
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
    hiredate_tab DateTabTyp;
    comm_tab RealTabTyp;
    indx BINARY_INTEGER;
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
    BEGIN ...
END;
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
    BEGIN ...
END;
BEGIN
    indx := 50;
    initialize(hiredate_tab, indx);  -- calls first version
    initialize(comm_tab, indx);      -- calls second version
    ...
END;
```

Restrictions on Overloading

Only local or packaged subprograms, or type methods, can be overloaded. Therefore, you cannot overload standalone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode. For example, you cannot overload the following two procedures:

```plsql
DECLARE
    ...
PROCEDURE reconcile (acct_no IN INTEGER) IS
    BEGIN ... END;
PROCEDURE reconcile (acct_no OUT INTEGER) IS
    BEGIN ... END;
```
You cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family. For instance, you cannot overload the following procedures because the datatypes INTEGER and REAL are in the same family:

```
DECLARE
  ...
  PROCEDURE charge_back (amount INTEGER) IS
    BEGIN ... END;
  PROCEDURE charge_back (amount REAL) IS
    BEGIN ... END;
```

Likewise, you cannot overload two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family. For example, you cannot overload the following procedures because the base types CHAR and LONG are in the same family:

```
DECLARE
  SUBTYPE Delimiter IS CHAR;
  SUBTYPE Text IS LONG;
  ...
  PROCEDURE scan (x Delimiter) IS
    BEGIN ... END;
  PROCEDURE scan (x Text) IS
    BEGIN ... END;
```

Finally, you cannot overload two functions that differ only in return type (the datatype of the return value) even if the types are in different families. For example, you cannot overload the following functions:

```
DECLARE
  ...
  FUNCTION acct_ok (acct_id INTEGER) RETURN BOOLEAN IS
    BEGIN ... END;
  FUNCTION acct_ok (acct_id INTEGER) RETURN INTEGER IS
    BEGIN ... END;
```
How Subprogram Calls Are Resolved

Figure 8–1 shows how the PL/SQL compiler resolves subprogram calls. When the compiler encounters a procedure or function 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 stops searching if 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. That is, they must match in number, order, and datatype (unless some formal parameters were assigned default values). If no match is found or if multiple matches are found, the compiler generates a semantic error.

In the following example, you call the enclosing procedure swap from within the function reconcile. However, the compiler generates an error because neither declaration of swap within the current scope matches the procedure call:

```sql
PROCEDURE swap (n1 NUMBER, n2 NUMBER) IS
    num1 NUMBER;
    num2 NUMBER;
    FUNCTION balance (...) RETURN REAL IS
        PROCEDURE swap (d1 DATE, d2 DATE) IS
            BEGIN
            ...
            END;
        PROCEDURE swap (b1 BOOLEAN, b2 BOOLEAN) IS
            BEGIN
            ...
            END;
        BEGIN
            ...
            swap(num1, num2);
        RETURN ...
        END balance;
        BEGIN
            ...
        END;
```
How Subprogram Calls Are Resolved

Figure 8–1  How the PL/SQL Compiler Resolves Calls

1. Encounter subprogram call
2. Compare name of called subprogram with names of any subprograms declared in current scope
3. Match(es) found?
   - Yes: Go to enclosing scope
   - No: Enclosing scope?
      - Yes: Compare actual parameter list in subprogram call with formal parameter list in subprogram declaration(s)
      - No: Match(es) found?
         - Yes: Multiple matches?
            - Yes: Resolve call
            - No: Generate syntax error
         - No: Resolve call
How Overloading Works with Inheritance

The overloading algorithm allows substituting a subtype value for a formal parameter that is a supertype. This capability is known as substitutability. If more than one instance of an overloaded procedure matches the procedure call, the following rules apply to determine which procedure is called:

If the only difference in the signatures of the overloaded procedures is that some parameters are object types from the same supertype-subtype hierarchy, the closest match is used. The closest match is one where all the parameters are at least as close as any other overloaded instance, as determined by the depth of inheritance between the subtype and supertype, and at least one parameter is closer.

A semantic error occurs when two overloaded instances match, and some argument types are closer in one overloaded procedure to the actual arguments than in any other instance.

A semantic error also occurs if some parameters are different in their position within the object type hierarchy, and other parameters are of different datatypes so that an implicit conversion would be necessary.

For example, here we create a type hierarchy with 3 levels:

```sql
CREATE TYPE super_t AS object
  (n NUMBER) NOT final;
CREATE OR replace TYPE sub_t under super_t
  (n2 NUMBER) NOT final;
CREATE OR replace TYPE final_t under sub_t
  (n3 NUMBER);
```

We declare two overloaded instances of a function, where the only difference in argument types is their position in this type hierarchy:

```sql
CREATE PACKAGE p IS
  FUNCTION foo (arg super_t) RETURN NUMBER;
  FUNCTION foo (arg sub_t) RETURN NUMBER;
END;
/
CREATE PACKAGE BODY p IS
  FUNCTION foo (arg super_t) RETURN NUMBER IS BEGIN RETURN 1; END;
  FUNCTION foo (arg sub_t) RETURN NUMBER IS BEGIN RETURN 2; END;
END;
/```
We declare a variable of type `final_t`, then call the overloaded function. The instance of the function that is executed is the one that accepts a `sub_t` parameter, because that type is closer to `final_t` in the hierarchy than `super_t` is.

```sql
SQL> set serveroutput on
SQL> declare
    v final_t := final_t(1,2,3);
begin
    dbms_output.put_line(p.foo(v));
end;
/
2
```

In the previous example, the choice of which instance to call is made at compile time. In the following example, this choice is made dynamically.

```sql
CREATE TYPE super_t2 AS object
  (n NUMBER, MEMBER FUNCTION foo RETURN NUMBER) NOT final;
/
CREATE TYPE BODY super_t2 AS
  MEMBER FUNCTION foo RETURN NUMBER IS BEGIN RETURN 1; END; END;
/
CREATE OR replace TYPE sub_t2 under super_t2
  (n2 NUMBER,
   OVERRIDING MEMBER FUNCTION foo RETURN NUMBER) NOT final;
/
CREATE TYPE BODY sub_t2 AS
  OVERRIDING MEMBER FUNCTION foo RETURN NUMBER IS BEGIN RETURN 2;
  END;
  END;
/
CREATE OR replace TYPE final_t2 under sub_t2
  (n3 NUMBER);
/
```

We declare `v` as an instance of `super_t2`, but because we assign a value of `sub_t2` to it, the appropriate instance of the function is called. This feature is known as **dynamic dispatch**.

```sql
SQL> set serveroutput on

declare
    v super_t2 := final_t2(1,2,3);
begin
    dbms_output.put_line(v.foo);
```
Accepting and Returning Multiple Rows with Table Functions

This section describes table functions. It also explains the generic datatypes ANYTYPE, ANYDATA, and ANYDATASET, which are likely to be used with table functions.

Major topics covered are:

- Overview of Table Functions
- Writing a Pipelined Table Function
- Parallelizing Table Functions
- How Table Functions Stream their Input Data

Overview of Table Functions

Table functions are functions that produce a collection of rows (either a nested table or a varray) that can be queried like a physical database table. You use a table function like the name of a database table, in the FROM clause of a query.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type (such as a VARRAY or a PL/SQL table) or a REF CURSOR.

Execution of a table function can be parallelized, and returned rows can be streamed directly to the next process without intermediate staging. Rows from a collection returned by a table function can also be pipelined—that is, iteratively returned as they are produced instead of in a batch after all processing of the table function’s input is completed.

Streaming, pipelining, and parallel execution of table functions can improve performance:

- By enabling multi-threaded, concurrent execution of table functions
- By eliminating intermediate staging between processes
- By improving query response time: With non-pipelined table functions, the entire collection returned by a table function must be constructed and returned to the server before the query can return a single result row. Pipelining enables
rows to be returned iteratively, as they are produced. This also reduces the memory that a table function requires, as the object cache does not need to materialize the entire collection.

- By iteratively providing result rows from the collection returned by a table function as the rows are produced instead of waiting until the entire collection is staged in tables or memory and then returning the entire collection.

Figure 8–2 shows a typical data-processing scenario in which data goes through several (in this case, three) transformations, implemented by table functions, before finally being loaded into a database. In this scenario, the table functions are not parallelized, and the entire result collection must be staged using a temporary table after each transformation.

Figure 8–2 Typical Data Processing with Unparallelized, Unpipelined Table Functions

By contrast, Figure 8–3, below, shows how streaming and parallel execution can streamline the same scenario.

Figure 8–3 Data Processing Using Pipelining and Parallel Execution

What Are Table Functions?

Table functions return a collection type instance representing rows in a table, and can be queried like a table by calling the function in the FROM clause of a query. Table functions use the TABLE keyword.
Accepting and Returning Multiple Rows with Table Functions

The following example shows a table function GetBooks that takes a CLOB as input and returns an instance of the collection type BookSet_t. The CLOB column stores a catalog listing of books in some format (either proprietary or following a standard such as XML). The table function returns all the catalogs and their corresponding book listings.

The collection type BookSet_t is defined as:

```sql
CREATE TYPE Book_t AS OBJECT
    ( name VARCHAR2(100),
      author VARCHAR2(30),
      abstract VARCHAR2(1000));

CREATE TYPE BookSet_t AS TABLE OF Book_t;
```

The CLOBs are stored in a table Catalogs:

```sql
CREATE TABLE Catalogs
    ( name VARCHAR2(30),
      cat CLOB);
```

Function GetBooks is defined as follows:

```sql
CREATE FUNCTION GetBooks(a CLOB) RETURN BookSet_t;
```

The query below returns all the catalogs and their corresponding book listings.

```sql
SELECT c.name, Book.name, Book.author, Book.abstract
    FROM Catalogs c,
    TABLE(GetBooks(c.cat)) Book;
```

What Are Pipelined Table Functions?

Data is said to be pipelined if it is consumed by a consumer (transformation) as soon as the producer (transformation) produces it, without being staged in tables or a cache before being input to the next transformation.

Pipelining enables a table function to return rows faster and can reduce the memory required to cache a table function’s results.

A pipelined table function can return the table function’s result collection in subsets. The returned collection behaves like a stream that can be fetched from on demand. This makes it possible to use a table function like a virtual table.

Pipelined table functions can be implemented in two ways:
Native PL/SQL approach: The consumer and producers can run on separate execution threads (either in the same or different process context) and communicate through a pipe or queuing mechanism. This approach is similar to co-routine execution.

Interface approach: The consumer and producers run on the same execution thread. Producer explicitly returns the control back to the consumer after producing a set of results. In addition, the producer caches the current state so that it can resume where it left off when the consumer invokes it again. The interface approach requires you to implement a set of well-defined interfaces in a procedural language. For details on the interface approach, see the Data Cartridges User’s Guide.

In the rest of this chapter, the term table function is used to refer to a pipelined table function—that is, a table function that returns a collection in an iterative, pipelined way.

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 (using either the interface approach or the native PL/SQL approach).

For example, the following code sketches the declarations that define a StockPivot function. This function converts a row of the type (Ticker, OpenPrice, ClosePrice) into two rows of the form (Ticker, PriceType, Price). Calling StockPivot for the row ("ORCL", 41, 42) generates two rows: ("ORCL", "O", 41) and ("ORCL", "C", 42).

Input data for the table function might come from a source such as table StockTable:

```
CREATE TABLE StockTable (
    ticker VARCHAR(4),
    open_price NUMBER,
    close_price NUMBER
);
```

Here are the declarations. See "PIPELINED IS...;" on page 8-35 for the function bodies.

```sql
-- Create the types for the table function's output collection
-- and collection elements
```
CREATE TYPE TickerType AS OBJECT
    (
        ticker VARCHAR2(4),
        PriceType VARCHAR2(1),
        price NUMBER
    );

CREATE TYPE TickerTypeSet AS TABLE OF TickerType;

-- Define the ref cursor type

CREATE PACKAGE refcur_pkg IS
    TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
/

-- Create the table function

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet
PIPELINED ... ;
/

Here is an example of a query that uses the StockPivot table function:

SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));

In the query above, the pipelined table function StockPivot fetches rows from the CURSOR subquery SELECT * FROM StockTable, performs the transformation, and pipelines the results back to the user as a table. The function produces two output rows (collection elements) for each input row.

Note that when a CURSOR subquery is passed from SQL to a REF CURSOR function argument as in the example above, the referenced cursor is already open when the function begins executing.

Writing a Pipelined Table Function

You declare a pipelined table function by specifying the PIPELINED keyword. This keyword indicates that the function will return rows iteratively. The return type of the pipelined table function must be a collection type, such as a nested table or a varray. You can declare this collection at the schema level, or inside a package. Inside the function, you return individual elements of the collection type.
For example, here are declarations for two pipelined table functions. (The function bodies are shown in later examples.)

```plsql
CREATE FUNCTION GetBooks(cat CLOB) RETURN BookSet_t PIPELINED IS ...;

CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet PIPELINED IS...
```

### Returning Results from Table Functions

In PL/SQL, the `PIPE ROW` statement causes a 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 runtime system provides the rows to the consumer in batches.) For example:

```plsql
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet PIPELINED IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
  in_rec p%ROWTYPE;
BEGIN
  LOOP
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
    -- first row
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.OpenPrice;
    PIPE ROW(out_rec);
    -- second row
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.ClosePrice;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
RETURN;
END;
/
```

In the example, the `PIPE ROW(out_rec)` statement pipelines data out of the PL/SQL table function. `out_rec` is a record, and its type matches the type of an element of the output collection.
接受和返回多行的表函数

管道行（PIPE ROW）语句只能用于管道化表函数的主体中；如果在任何其他地方使用，会引发错误。PIPE ROW 语句可以省略对于不返回任何行的管道化表函数。

管道化表函数必须有一个 RETURN 语句，该语句不返回任何值。RETURN 语句将控制权转回消费者，并确保下一次fetch会引发NO_DATA_FOUND异常。

Oracle具有三种特殊的SQL数据类型，它们允许您动态封装和访问类型描述、数据实例以及任何其他SQL类型的任意数据实例。您还可以使用这三个特殊类型创建匿名（即未命名）类型，包括匿名集合类型。这些类型是SYS.ANYTYPE，SYS.ANYDATA，和SYS.ANYDATASET。SYS.ANYDATA类型在某些情况下可以作为表函数的返回值。

**See Also:** Oracle9i Supplied PL/SQL Packages and Types Reference

对于ANYTYPE，ANYDATA，和ANYDATASET类型的接口信息，以及DBMS_TYPES包的使用。

### 跨PL/SQL表函数管道数据

在串行执行中，结果通过使用类似共用体执行的类似方法从一个PL/SQL表函数传递到另一个。例如，以下语句将结果从函数g传递到函数f：

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));
```

并行执行类似，但每个函数执行在不同的进程（或一系列进程）。

### 查询表函数

管道化的表函数用于SELECT语句的FROM子句。结果行由Oracle迭代从表函数实现。例如：

```
SELECT x.Ticker, x.Price
FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable))) x
WHERE x.PriceType='C';
```
Optimizing Multiple Calls to Table Functions

Multiple invocations of a 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());
```

However, if the output of a table function is determined solely by the values passed into it as arguments, such that the function always produces exactly the same result value for each respective combination of values passed in, you can declare the function `DETERMINISTIC`, and Oracle will automatically buffer rows for it. Note, though, that the database has no way of knowing whether a function marked `DETERMINISTIC` really is `DETERMINISTIC`, and if one is not, results will be unpredictable.

Fetching from the Results of 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
BEGIN
    OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
    SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
END;
```
does not execute as well as:

```
SELECT * FROM TABLE(g(CURSOR(SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab))))));
```

This is so even ignoring the overhead associated with executing two SQL statements and assuming that the results can be pipelined between the two statements.

### Passing Data with Cursor Variables

You can pass a set of rows to a PL/SQL function in a `REF CURSOR` parameter. For example, this function is declared to accept an argument of the predefined weakly typed `REF CURSOR` type `SYS_REFCURSOR`:

```plsql
FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;
```

Results of a subquery can be passed to a function directly:

```
SELECT * FROM TABLE(f(CURSOR(SELECT empno FROM tab)));
```

In the example above, the `CURSOR` keyword is required to indicate that the results of a subquery should be passed as a `REF CURSOR` parameter.

A predefined weak `REF CURSOR` type `SYS_REFCURSOR` is also supported. With `SYS_REFCURSOR`, you do not need to first create a `REF CURSOR` type in a package before you can use it.

To use a strong `REF CURSOR` type, you still must create a PL/SQL package and declare a strong `REF CURSOR` type in it. Also, if you are using a strong `REF CURSOR` type as an argument to a table function, then the actual type of the `REF CURSOR` argument must match the column type, or an error is generated. Weak `REF CURSOR` arguments to table functions can only be partitioned using the `PARTITION BY ANY` clause. You cannot use range or hash partitioning for weak `REF CURSOR` arguments.

### Using Multiple REF CURSOR Input Variables

PL/SQL functions can accept multiple `REF CURSOR` input variables:

```plsql
CREATE FUNCTION g(p1 pkg.refcur_t1, p2 pkg.refcur_t2) RETURN...
   PIPELINED ... ;
```

Function `g` can be invoked as follows:
SELECT * FROM TABLE(g(CURSOR(SELECT empno FROM tab),
CURSOR(SELECT * FROM emp));

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(...)))));

**Explicitly Opening a REF CURSOR for a Query**

You can explicitly open a REF CURSOR for a query and pass it as a parameter to a table function:

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 should not explicitly try to close the cursor.

**Performing DML Operations Inside Table Functions**

To execute DML statements, a table function must be declared with the autonomous transaction pragma. This pragma causes the function to execute in an autonomous transaction not shared by other processes.

Use the following syntax to declare a table function with the autonomous transaction pragma:

CREATE FUNCTION f(p SYS_REFCURSOR) return CollType PIPELINED IS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN ... END;

During parallel execution, each instance of the table function creates an independent transaction.

**Performing DML Operations on Table Functions**

Table functions cannot be the target table in UPDATE, INSERT, or DELETE statements. For example, the following statements will raise an error:

UPDATE F(CURSOR(SELECT * FROM tab)) SET col = value;
However, you can create a view over a table function and use INSTEAD OF triggers to update it. For example:

```sql
CREATE VIEW BookTable AS
    SELECT x.Name, x.Author
    FROM TABLE(GetBooks('data.txt')) x;
```

The following INSTEAD OF trigger is fired when the user inserts a row into the BookTable view:

```sql
CREATE TRIGGER BookTable_insert
    INSTEAD OF INSERT ON BookTable
    REFERENCING NEW AS n
    FOR EACH ROW
    BEGIN
        ...
        END;
```

INSTEAD OF triggers can be defined for all DML operations on a view built on a table function.

### Handling Exceptions in Table Functions

Exception handling in table functions works just as it does with ordinary user-defined 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.

### Parallelizing Table Functions

For a table function to be executed in parallel, it must have a partitioned input parameter. Parallelism is turned on for a table function if, and only if, both the following conditions are met:

- The function has a PARALLEL_ENABLE clause in its declaration
Exactly one REF CURSOR argument is specified with a PARTITION BY clause. If the PARTITION BY clause is not specified for any input REF CURSOR as part of the PARALLEL_ENABLE clause, the SQL compiler cannot determine how to partition the data correctly. Note that only strongly typed REF CURSOR arguments can be specified in the PARTITION BY clause, unless you use PARTITION BY ANY.

Parallel Execution of Table Functions

With parallel execution of a function that appears in the SELECT list, execution of the function is pushed down to and conducted by multiple slave scan processes. These each execute the function on a segment of the function’s input data.

For example, the query

```sql
SELECT f(col1) FROM tab;
```

is parallelized if `f` is a pure function. The SQL executed by a slave scan process is similar to:

```sql
SELECT f(col1) FROM tab WHERE ROWID BETWEEN :b1 AND :b2;
```

Each slave scan operates on a range of rowids and applies function `f` to each contained row. Function `f` is then executed by the scan processes; it does not run independently of them.

Unlike a function that appears in the SELECT list, a table function is called in the FROM clause and returns a collection. This affects the way that table function input data is partitioned among slave scans because the partitioning approach must be appropriate for the operation that the table function performs. (For example, an ORDER BY operation requires input to be range-partitioned, whereas a GROUP BY operation requires input to be hash partitioned.)

A table function itself specifies in its declaration the partitioning approach that is appropriate for it. (See “Input Data Partitioning” on page 8-42.) The function is then executed in a two-stage operation. First, one set of slave processes partitions the data as directed in the function’s declaration; then a second set of slave processes executes the table function in parallel on the partitioned data.

For example, the table function in the following query has a REF CURSOR parameter:

```sql
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
```
Parallelizing Table Functions

The scan is performed by one set of slave processes, which redistributes the rows (based on the partitioning method specified in the function declaration) to a second set of slave processes that actually executes function $f$ in parallel.

Input Data Partitioning

The table function declaration can specify data partitioning for exactly one `REF CURSOR` parameter. The syntax to do this is as follows:

```sql
CREATE FUNCTION f(p ref cursor type) RETURN rec_tab_type PIPELINED
   PARALLEL_ENABLE (PARTITION p BY [\{HASH | RANGE\} (column list) | ANY }] IS
BEGIN ... END;
```

The `PARTITION...BY` phrase in the `PARALLEL_ENABLE` clause specifies which one of the input cursors to partition and what columns to use for partitioning.

When explicit column names are specified in the column list, the partitioning method can be `RANGE` or `HASH`. The input rows will be hash- or range-partitioned on the columns specified.

The `ANY` keyword indicates that the function behavior is independent of the partitioning of the input data. When this keyword is used, the runtime system randomly partitions the data among the slaves. This keyword is appropriate for use with functions that take in one row, manipulate its columns, and generate output row(s) based on the columns of this row only.

For example, the pivot-like function `StockPivot` shown below takes as input a row of the type:

(Ticker varchar(4), OpenPrice number, ClosePrice number)

and generates rows of the type:

(Ticker varchar(4), PriceType varchar(1), Price number).

So the row ("ORCL", 41, 42) generates two rows ("ORCL", "O", 41) and ("ORCL", "C", 42).

```sql
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN rec_tab_type PIPELINED
   PARALLEL_ENABLE (PARTITION p BY ANY) IS
   ret_rec rec_type;
BEGIN
   FOR rec IN p LOOP
      ret_rec.Ticker := rec.Ticker;
      ret_rec.PriceType := "O";
      ret_rec.Price := rec.OpenPrice;
   END LOOP;
```

8-42  PL/SQL User’s Guide and Reference
PIPE ROW(ret_rec);

ret_rec.Ticker := rec.Ticker;  -- Redundant; not required
ret_rec.PriceType := "C";
ret_rec.Price := rec.ClosePrice;
push ret_rec;
END LOOP;
RETURN;
END;

The function $f$ can be used to generate another table from Stocks table in the following manner:

```
INSERT INTO AlternateStockTable
SELECT * FROM
TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

If the StockTable is scanned in parallel and partitioned on OpenPrice, then the function StockPivot is combined with the data-flow operator doing the scan of StockTable and thus sees the same partitioning.

If, on the other hand, the StockTable is not partitioned, and the scan on it does not execute in parallel, the insert into AlternateStockTable also runs sequentially. Here is a slightly more complex example:

```
INSERT INTO AlternateStockTable
SELECT *
FROM TABLE(f(CURSOR(SELECT * FROM Stocks)),
           TABLE(g(CURSOR( ... ))))
WHERE join_condition;
```

where $g$ is defined to be:

```
CREATE FUNCTION g(p refcur_pkg.refcur_t) RETURN ... PIPELINED
   PARALLEL_ENABLE (PARTITION p BY ANY)
BEGIN ... END;
```

If function $g$ runs in parallel and is partitioned by ANY, then the parallel insert can belong in the same data-flow operator as $g$.

Whenever the ANY keyword is specified, the data is partitioned randomly among the slaves. This effectively means that the function is executed in the same slave set which does the scan associated with the input parameter.
No redistribution or repartitioning of the data is required here. If the cursor p itself is not parallelized, the incoming data is randomly partitioned on the columns in the column list. The round-robin table queue is used for this partitioning.

**Parallel Execution of Leaf-level Table Functions**

To use parallel execution with a function that produces multiple rows, but does not need to accept multiple rows as input and so does not require a `REF CURSOR`, arrange things so as to create a need for a `REF CURSOR`. That way, the function will have some way to partition the work.

For example, suppose that you want a function to read a set of external files in parallel and return the records they contain. To provide work for a `REF CURSOR`, you might first create a table and populate it with the filenames. A `REF CURSOR` over this table can then be passed as a parameter to the table function (readfiles). The following code shows how this might be done:

```sql
CREATE TABLE filetab(filename VARCHAR(20));
INSERT INTO filetab VALUES('file0');
INSERT INTO filetab VALUES('file1');
...
INSERT INTO filetab VALUES('fileN');
SELECT * FROM (SELECT filename FROM filetab);
CREATE FUNCTION readfiles(p pkg.rc_t) RETURN coll_type PARALLEL_ENABLE(PARTITION p BY ANY) IS
  ret_rec rec_type;
BEGIN
  FOR rec IN p LOOP
    done := FALSE;
    WHILE (done = FALSE) LOOP
      done := readfilerecord(rec.filename, ret_rec);
      PIPE ROW(ret_rec);
    END LOOP;
  END LOOP;
RETURN;
END;
```

```sql
SELECT * FROM (SELECT filename FROM filetab) READFILES(CURSOR(SELECT filename FROM filetab));
```
How Table Functions Stream their Input Data

The way in which a table function orders or clusters rows that it fetches from cursor arguments is called **data streaming**. A function can stream its input data in any of the following ways:

- Place no restriction on the ordering of the incoming rows
- Order them on a particular key column or columns
- Cluster them on a particular key

**Clustering** causes rows that have the same key values to appear together but does not otherwise do any ordering of rows.

You control the behavior of the input stream using the following syntax:

```plsql
FUNCTION f(p ref_cursor_type) RETURN tab_rec_type [PIPELINED]
  {[ORDER | CLUSTER] BY column_list}
  PARALLEL_ENABLE({PARTITION p BY
  {ANY | (HASH | RANGE) column_list})
IS
BEGIN ... END;
```

Input streaming may be specified for either sequential or parallel execution of a function.

If an `ORDER BY` or `CLUSTER BY` clause is not specified, rows are input in a (random) order.

---

**Note:** The semantics of `ORDER BY` are different for parallel execution from the semantics of the `ORDER BY` clause in a SQL statement. In a SQL statement, the `ORDER BY` clause globally orders the entire data set. In a table function, the `ORDER BY` clause orders the respective rows local to each instance of the table function running on a slave.

---

The following example illustrates the syntax for ordering the input stream. In the example, function `f` takes in rows of the kind `(Region, Sales)` and returns rows of the form `(Region, AvgSales)`, showing average sales for each region.

```plsql
CREATE FUNCTION f(p ref_cursor_type) RETURN tab_rec_type PIPELINED
  CLUSTER BY Region
  PARALLEL_ENABLE(PARTITION p BY Region) IS
  ret_rec rec_type;
```
cnt number;
sum number;
BEGIN
  FOR rec IN p LOOP
    IF (first rec in the group) THEN
      cnt := 1;
      sum := rec.Sales;
    ELSIF (last rec in the group) THEN
      IF (cnt <> 0) THEN
        ret_rec.Region := rec.Region;
        ret_rec.AvgSales := sum/cnt;
        PIPE ROW(ret_rec);
      END IF;
    ELSE
      cnt := cnt + 1;
      sum := sum + rec.Sales;
    END IF;
  END LOOP;
RETURN;
END;

Choosing Between Partitioning and Clustering for Parallel Execution

Partitioning and clustering are easily confused, but they do different things. For example, sometimes partitioning can be sufficient without clustering in parallel execution.

Consider a function SmallAggr that performs in-memory aggregation of salary for each department_id, where department_id can be either 1, 2, or 3. The input rows to the function can be partitioned by HASH on department_id such that all rows with department_id equal to 1 go to one slave, all rows with department_id equal to 2 go to another slave, and so on.

The input rows do not need to be clustered on department_id to perform the aggregation in the function. Each slave could have a 1x3 array SmallSum[1..3] in which the aggregate sum for each department_id is added in memory into SmallSum[department_id]. On the other hand, if the number of unique values of department_id were very large, you would want to use clustering to compute department aggregates and write them to disk one department_id at a time.
Invoker Rights Versus Definer Rights

By default, stored procedures and SQL methods execute with the privileges of their owner, not their current user. Such definer-rights subprograms are bound to the schema in which they reside. For example, assume that dept tables reside in schemas scott and blake, and that the following standalone procedure resides in schema scott:

```sql
CREATE PROCEDURE create_dept (
    my_deptno NUMBER,
    my_dname  VARCHAR2,
    my_loc    VARCHAR2) AS
BEGIN
    INSERT INTO dept VALUES (my_deptno, my_dname, my_loc);
END;
```

Also assume that user scott has granted the EXECUTE privilege on this procedure to user blake. When user blake calls the procedure, the INSERT statement executes with the privileges of user scott. Also, the unqualified references to table dept is resolved in schema scott. So, even though user blake called the procedure, it updates the dept table in schema scott.

How can subprograms in one schema manipulate objects in another schema? One way is to fully qualify references to the objects, as in

```sql
INSERT INTO blake.dept ...
```

However, that hampers portability. As a workaround, you can define the schema name as a variable in SQL*Plus.

Another way is to copy the subprograms into the other schema. However, that hampers maintenance.

A better way is to use the AUTHID clause, which enables stored procedures and SQL methods to execute with the privileges and schema context of their current user.

Such invoker-rights subprograms are not bound to a particular schema. They can be run by a variety of users. The following version of procedure create_dept executes with the privileges of its current user and inserts rows into the dept table in that user’s schema:

```sql
CREATE PROCEDURE create_dept (
    my_deptno NUMBER,
    my_dname  VARCHAR2,
    my_loc    VARCHAR2) AUTHID CURRENT_USER AS
BEGIN
```
INSERT INTO dept VALUES (my_deptno, my_dname, my_loc);
END;

Advantages of Invoker Rights

Invoker-rights subprograms let you reuse code and centralize application logic. They are especially useful in applications that store data in different schemas. In such cases, multiple users can manage their own data using a single code base.

Consider a company that uses a definer-rights (DR) procedure to analyze sales. To provide local sales statistics, procedure analyze must access sales tables that reside at each regional site. So, as Figure 8–4 shows, the procedure must also reside at each regional site. This causes a maintenance problem.

Figure 8–4 Definer-Rights Problem: Multiple Copies of Identical Procedures

To solve the problem, the company installs an invoker-rights (IR) version of procedure analyze at headquarters. Now, as Figure 8–5 shows, all regional sites can use the same procedure to query their own sales tables.
To restrict access to sensitive data, you can have an invoker-rights subprogram call a definer-rights subprogram. Suppose headquarters would like procedure `analyze` to calculate sales commissions and update a central `payroll` table.

That presents a problem because current users of `analyze` should not have direct access to the `payroll` table, which stores employee salaries and other sensitive data. As Figure 8–6 shows, the solution is to have procedure `analyze` call definer-rights procedure `calc_comm`, which in turn updates the `payroll` table.
Specifying the Privileges for a Subprogram with the AUTHID Clause

To implement invoker rights, use the AUTHID clause, which specifies whether a subprogram executes with the privileges of its owner or its current user. It also specifies whether external references (that is, references to objects outside the subprogram) are resolved in the schema of the owner or the current user.

The AUTHID clause is allowed only in the header of a standalone subprogram, a package spec, or an object type spec. The header syntax is

-- standalone function
CREATE [OR REPLACE] FUNCTION [schema_name.]function_name
[(parameter_list)] RETURN datatype
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- standalone procedure
CREATE [OR REPLACE] PROCEDURE [schema_name.]procedure_name
[(parameter_list)]
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- package spec
CREATE [OR REPLACE] PACKAGE [schema_name.]package_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- object type spec
CREATE [OR REPLACE] TYPE [schema_name.]object_type_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT
where DEFINER is the default option. In a package or object type, the AUTHID clause applies to all subprograms.

**Note:** Most supplied PL/SQL packages (such as DBMS_LOB, DBMS_PIPE, DBMS_ROWID, DBMS_SQL, and UTL_REF) are invoker-rights packages.

### Who Is the Current User During Subprogram Execution?

In a sequence of calls, if an invoker-rights subprogram is the first subprogram called, the current user is the session user. That remains true until a definer-rights subprogram is called, in which case the owner of that subprogram becomes the current user. If the definer-rights subprogram calls any invoker-rights subprograms, they execute with the privileges of the owner. When the definer-rights subprogram exits, control reverts to the previous current user.

### How External References Are Resolved in Invoker-Rights Subprograms

If you specify AUTHID CURRENT_USER, the privileges of the current user are checked at run time, and external references are resolved in the schema of the current user. However, this applies only to external references in:

- SELECT, INSERT, UPDATE, and DELETE data manipulation statements
- The LOCK TABLE transaction control statement
- OPEN and OPEN-FOR cursor control statements
- EXECUTE IMMEDIATE and OPEN-FOR-USING dynamic SQL statements
- SQL statements parsed using DBMS_SQL.PARSE()

For all other statements, the privileges of the owner are checked at compile time, and external references are resolved in the schema of the owner. For example, the assignment statement below refers to the packaged function `balance`. This external reference is resolved in the schema of the owner of procedure `reconcile`.

```sql
CREATE PROCEDURE reconcile (acc_id IN INTEGER)
AUTHID CURRENT_USER AS
  bal NUMBER;
BEGIN
  bal := bank_ops.balance(acct_id);
  ...
END;
```
The Need for Template Objects in Invoker-Rights Subprograms

External references in an invoker-rights subprogram are resolved in the schema of the current user at run time. However, the PL/SQL compiler must resolve all references at compile time. So, the owner must create template objects in his or her schema beforehand. At run time, the template objects and the actual objects must match. Otherwise, you get an error or unexpected results.

For example, suppose user scott creates the following database table and standalone procedure:

```sql
CREATE TABLE emp (  
    empno NUMBER(4),  
    ename VARCHAR2(15));
/
CREATE PROCEDURE evaluate (my_empno NUMBER)  
    AUTHID CURRENT_USER AS  
    my_ename VARCHAR2(15);  
BEGIN  
    SELECT ename INTO my_ename FROM emp WHERE empno = my_empno;  
    ...  
END;
/
```

Now, suppose user blake creates a similar database table, then calls procedure evaluate, as follows:

```sql
CREATE TABLE emp (  
    empno NUMBER(4),  
    ename VARCHAR2(15),  
    my_empno NUMBER(4)); -- note extra column
/
DECLARE  
    ...  
BEGIN  
    ...  
    scott.evaluate(7788);  
END;
/
```

The procedure call executes without error but ignores column my_empno in the table created by user blake. That is because the actual table in the schema of the current user does not match the template table used at compile time.
Overriding Default Name Resolution in Invoker-Rights Subprograms

Occasionally, you might want to override the default invoker-rights behavior. Suppose user scott defines the standalone procedure below. Notice that the SELECT statement calls an external function. Normally, this external reference would be resolved in the schema of the current user.

```
CREATE PROCEDURE calc_bonus (emp_id INTEGER) 
  AUTHID CURRENT_USER AS
  mid_sal REAL;
BEGIN
  SELECT median(sal) INTO mid_sal FROM emp;
  ... 
END;
```

To have the reference resolved in the schema of the owner, create a public synonym for the function using the CREATE SYNONYM statement, as follows:

```
CREATE PUBLIC SYNONYM median FOR scott.median;
```

This works unless the current user has defined a function or private synonym named median. Alternatively, you can fully qualify the reference, as follows:

```
BEGIN
  SELECT scott.median(sal) INTO mid_sal FROM emp;
  ... 
END;
```

This works unless the current user has defined a package named scott that contains a function named median.

Granting Privileges on Invoker-Rights Subprograms

To call a subprogram directly, users must have the EXECUTE privilege on that subprogram. By granting the privilege, you allow a user to:

- Call the subprogram directly
- Compile functions and procedures that call the subprogram

For external references resolved in the current user’s schema (such as those in DML statements), the current user must have the privileges needed to access schema objects referenced by the subprogram. For all other external references (such as function calls), the owner’s privileges are checked at compile time, and no run-time check is done.
A definer-rights subprogram operates under the security domain of its owner, no matter who is executing it. So, the owner must have the privileges needed to access schema objects referenced by the subprogram.

You can write a program consisting of multiple subprograms, some with definer rights and others with invoker rights. Then, you can use the EXECUTE privilege to restrict program entry points (controlled step-in). That way, users of an entry-point subprogram can execute the other subprograms indirectly but not directly.

**Granting Privileges on an Invoker-Rights Subprogram: Example**

Suppose user `util` grants the EXECUTE privilege on subprogram `fft` to user `app`, as follows:

```sql
GRANT EXECUTE ON util.fft TO app;
```

Now, user `app` can compile functions and procedures that call subprogram `fft`. At run time, no privilege checks on the calls are done. So, as Figure 8–7 on page 8-55 shows, user `util` need not grant the EXECUTE privilege to every user who might call `fft` indirectly.

Notice that subprogram `util.fft` is called directly only from invoker-rights subprogram `app.entry`. So, user `util` must grant the EXECUTE privilege only to user `app`. When `util.fft` is executed, its current user could be `app`, `scott`, or `blake` even though `scott` and `blake` were not granted the EXECUTE privilege.
Using Roles with Invoker-Rights Subprograms

The use of roles in a subprogram depends on whether it executes with definer rights or invoker rights. Within a definer-rights subprogram, all roles are disabled. Roles are not used for privilege checking, and you cannot set roles.

Within an invoker-rights subprogram, roles are enabled (unless the subprogram was called directly or indirectly by a definer-rights subprogram). Roles are used for privilege checking, and you can use native dynamic SQL to set roles for the session. However, you cannot use roles to grant privileges on template objects because roles apply at run time, not at compile time.

Using Views and Database Triggers with Invoker-Rights Subprograms

For invoker-rights subprograms executed within a view expression, the view owner, not the view user, is considered to be the current user. For example, suppose user scott creates a view as shown below. The invoker-rights function layout always executes with the privileges of user scott, who is the view owner.

```
CREATE VIEW payroll AS SELECT layout (3) FROM dual;
```

This rule also applies to database triggers.
Using Database Links with Invoker-Rights Subprograms

Invoker rights affect only one kind of database link—current-user links, which are created as follows:

```sql
CREATE DATABASE LINK link_name CONNECT TO CURRENT_USER
    USING connect_string;
```

A current-user link lets you connect to a remote database as another user, with that user’s privileges. To connect, Oracle uses the username of the current user (who must be a global user). Suppose an invoker-rights subprogram owned by user blake references the database link below. If global user scott calls the subprogram, it connects to the Dallas database as user scott, who is the current user.

```sql
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER USING ...
```

If it were a definer-rights subprogram, the current user would be blake. So, the subprogram would connect to the Dallas database as global user blake.

Using Object Types with Invoker-Rights Subprograms

To define object types for use in any schema, specify the AUTHID CURRENT_USER clause. (For more information about object types, see Chapter 10.) Suppose user blake creates the following object type:

```sql
CREATE TYPE Num AUTHID CURRENT_USER AS OBJECT (x NUMBER,
    STATIC PROCEDURE new_num (n NUMBER, schema_name VARCHAR2, table_name VARCHAR2))
);

CREATE TYPE BODY Num AS
    STATIC PROCEDURE new_num (n NUMBER, schema_name VARCHAR2, table_name VARCHAR2) IS
        sql_stmt VARCHAR2(200);
    BEGIN
        sql_stmt := 'INSERT INTO ' || schema_name || ' . ' || table_name || ' VALUES (blake.Num(:1))';
        EXECUTE IMMEDIATE sql_stmt USING n;
    END;
END;
```

Then, user blake grants the EXECUTE privilege on object type Num to user scott:

```sql
GRANT EXECUTE ON Num TO scott;
```
Finally, user scott creates an object table to store objects of type Num, then calls procedure new_num to populate the table:

```sql
CONNECT scott/tiger;
CREATE TABLE num_tab OF blake.Num;
/
BEGIN
  blake.Num.new_num(1001, 'scott', 'num_tab');
  blake.Num.new_num(1002, 'scott', 'num_tab');
  blake.Num.new_num(1003, 'scott', 'num_tab');
END;
/
```

The calls succeed because the procedure executes with the privileges of its current user (scott), not its owner (blake).

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.

### Calling Invoker-Rights Instance Methods

An invoker-rights instance method executes with the privileges of the invoker, not the creator of the instance. Suppose that Person is an invoker-rights object type, and that user scott creates p1, an object of type Person. If user blake calls instance method change_job to operate on object p1, the current user of the method is blake, not scott. Consider the following example:

```sql
-- user blake creates a definer-rights procedure
CREATE PROCEDURE reassign (p Person, new_job VARCHAR2) AS
BEGIN
  -- user blake calls method change_job, so the
  -- method executes with the privileges of blake
  p.change_job(new_job);
  ...
END;

-- user scott passes a Person object to the procedure
DECLARE
  p1 Person;
```
BEGIN
  p1 := Person(...);
  blake.reassign(p1, 'CLERK');
  ...
END;

Understanding and Using Recursion

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, ...), which was first used to model the growth of a rabbit colony, 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 $n$:

$$n! = n \times (n - 1)!$$

What Is a Recursive Subprogram?

A recursive subprogram is one that calls itself. Think of a recursive call as a call to some other subprogram that does the same task as your subprogram. 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. As a result, your program might exceed the limit set by the Oracle initialization parameter OPEN_CURSORS.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a terminating condition. Otherwise, the recursion would (theoretically) go on forever. In practice, if a recursive subprogram strays into infinite regress, PL/SQL eventually runs out of memory and raises the predefined exception STORAGE_ERROR.

Recursion Example: Computing the Factorial

To solve some programming problems, you must repeat a sequence of statements until a condition is met. You can use iteration or recursion to solve such problems.
Use recursion when the problem can be broken down into simpler versions of itself. For example, you can evaluate $3!$ as follows:

$$
\begin{align*}
0! & = 1 \quad \text{-- by definition} \\
1! & = 1 \times 0! = 1 \\
2! & = 2 \times 1! = 2 \\
3! & = 3 \times 2! = 6
\end{align*}
$$

To implement this algorithm, you might write the following recursive function, which returns the factorial of a positive integer:

```plsql
FUNCTION fac (n POSITIVE) RETURN INTEGER IS  -- returns n!
BEGIN
  IF n = 1 THEN  -- terminating condition
    RETURN 1;
  ELSE
    RETURN n \times \text{fac}(n - 1);  -- recursive call
  END IF;
END fac;
```

At each recursive call, $n$ is decremented. Eventually, $n$ becomes 1 and the recursion stops.

**Recursion Example: Traversing Tree-Structured Data**

Consider the procedure below, which finds the staff of a given manager. The procedure declares two formal parameters, `mgr_no` and `tier`, which represent the manager’s employee number and a tier in his or her departmental organization. Staff members reporting directly to the manager occupy the first tier.

```plsql
PROCEDURE find_staff (mgr_no NUMBER, tier NUMBER := 1) IS
  boss_name VARCHAR2(10);
  CURSOR c1 (boss_no NUMBER) IS
    SELECT empno, ename FROM emp WHERE mgr = boss_no;
  BEGIN
    /* Get manager’s name. */
    SELECT ename INTO boss_name FROM emp WHERE empno = mgr_no;
    IF tier = 1 THEN
      INSERT INTO staff  -- single-column output table
        VALUES (boss_name || ' manages the staff');
    END IF;
    /* Find staff members who report directly to manager. */
    FOR ee IN c1 (mgr_no) LOOP
      INSERT INTO staff
        VALUES (boss_name || ' manages ' || ee.ename
```
When called, the procedure accepts a value for `mgr_no` but uses the default value of `tier`. For example, you might call the procedure as follows:

```
find_staff(7839);
```

The procedure passes `mgr_no` to a cursor in a cursor `FOR` loop, which finds staff members at successively lower tiers in the organization. At each recursive call, a new instance of the `FOR` loop is created and another cursor is opened, but prior cursors stay positioned on the next row in their result sets.

When a fetch fails to return a row, the cursor is closed automatically and the `FOR` loop is exited. Since the recursive call is inside the `FOR` loop, the recursion stops.

Unlike the initial call, each recursive call passes a second actual parameter (the next tier) to the procedure.

**Tip: Perform Recursive Queries with the CONNECT BY Clause**

The last example illustrates recursion, not the efficient use of set-oriented SQL statements. You might want to compare the performance of the recursive procedure to that of the following SQL statement, which does the same task:

```
INSERT INTO staff
  SELECT PRIOR ename || ' manages ' || ename || ' on tier ' || TO_CHAR(LEVEL - 1)
  FROM emp
  START WITH empno = 7839
  CONNECT BY PRIOR empno = mgr;
```

The SQL statement is appreciably faster. However, the procedure is more flexible. For example, a multi-table query cannot contain the `CONNECT BY` clause. So, unlike the procedure, the SQL statement cannot be modified to do joins. (A *join* combines rows from two or more database tables.) In addition, a procedure can process data in ways that a single SQL statement cannot.
Using Mutual Recursion

Subprograms are *mutually recursive* if they directly or indirectly call each other. In the example below, the Boolean functions `odd` and `even`, which determine whether a number is odd or even, call each other directly. The forward declaration of `odd` is necessary because `even` calls `odd`, which is not yet declared when the call is made.

```plsql
FUNCTION odd (n NATURAL) RETURN BOOLEAN; -- forward declaration

FUNCTION even (n NATURAL) RETURN BOOLEAN IS
BEGIN
  IF n = 0 THEN
    RETURN TRUE;
  ELSE
    RETURN odd(n - 1); -- mutually recursive call
  END IF;
END even;

FUNCTION odd (n NATURAL) RETURN BOOLEAN IS
BEGIN
  IF n = 0 THEN
    RETURN FALSE;
  ELSE
    RETURN even(n - 1); -- mutually recursive call
  END IF;
END odd;
```

When a positive integer `n` is passed to `odd` or `even`, the functions call each other by turns. At each call, `n` is decremented. Ultimately, `n` becomes zero and the final call returns `TRUE` or `FALSE`. For instance, passing the number 4 to `odd` results in this sequence of calls:

- `odd(4)`
- `even(3)`
- `odd(2)`
- `even(1)`
- `odd(0)  -- returns FALSE`

On the other hand, passing the number 4 to `even` results in this sequence of calls:

- `even(4)`
- `odd(3)`
- `even(2)`
- `odd(1)`
- `even(0)  -- returns TRUE`
Recursion Versus Iteration

Unlike iteration, recursion is not essential to PL/SQL programming. Any problem that can be solved using recursion can be solved using iteration. Also, the iterative version of a subprogram is usually easier to design than the recursive version. However, the recursive version is usually simpler, smaller, and therefore easier to debug. Compare the following functions, which compute the \( n \)th Fibonacci number:

```plsql
-- recursive version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
  BEGIN
    IF (n = 1) OR (n = 2) THEN
      RETURN 1;
    ELSE
      RETURN fib(n - 1) + fib(n - 2);
    END IF;
  END fib;

-- iterative version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
  pos1 INTEGER := 1;
  pos2 INTEGER := 0;
  cum   INTEGER;
  BEGIN
    IF (n = 1) OR (n = 2) THEN
      RETURN 1;
    ELSE
      cum := pos1 + pos2;
      FOR i IN 3..n LOOP
        pos2 := pos1;
        pos1 := cum;
        cum := pos1 + pos2;
      END LOOP;
      RETURN cum;
    END IF;
  END fib;
```

The recursive version of \( \text{fib} \) is more elegant than the iterative version. However, the iterative version is more efficient; it runs faster and uses less storage. That is because each recursive call requires additional time and memory. As the number of recursive calls gets larger, so does the difference in efficiency. Still, if you expect the number of recursive calls to be small, you might choose the recursive version for its readability.
Calling External Subprograms

PL/SQL is a powerful development tool; you can use it for almost any purpose. However, it is specialized for SQL transaction processing. So, some tasks are more quickly done in a lower-level language such as C, which is very efficient at machine-precision calculations. Other tasks are more easily done in a fully object-oriented, standardized language such as Java.

To support such special-purpose processing, you can use PL/SQL call specs to invoke external subprograms (that is, subprograms written in other languages). This makes the strengths and capabilities of those languages available to you from PL/SQL.

For example, you can call Java stored procedures from any PL/SQL block, subprogram, or package. Suppose you store the following Java class in the database:

```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 emp SET sal = sal * ? WHERE empno = ?";
        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 this call spec:

```sql
CREATE PROCEDURE raise_salary (empno NUMBER, pct NUMBER)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
```
Later, you might call procedure `raise_salary` from an anonymous PL/SQL block, as follows:

```plsql
DECLARE
  emp_id  NUMBER;
  percent NUMBER;
BEGIN
  -- get values for emp_id and percent
  raise_salary(emp_id, percent);  -- call external subprogram
END;
```

Typically, 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 let you extend the functionality of the database server, and move computation-bound programs from client to server, where they execute faster.

For more information about Java stored procedures, see *Oracle9i Java Stored Procedures Developer's Guide*. For more information about external C subprograms, see *Oracle9i Application Developer's Guide - Fundamentals*.

---

**Creating Dynamic Web Pages with PL/SQL Server Pages**

PL/SQL Server Pages (PSPs) enable you to develop Web pages with dynamic content. They are an alternative to coding a stored procedure 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 procedures.

For more information about creating and using PSPs, see *Oracle9i Application Developer's Guide - Fundamentals*.
Goods which are not shared are not goods. —Fernando de Rojas

This chapter shows you how to bundle related PL/SQL programming resources into a package. The resources might include a collection of procedures or a pool of type definitions and variable declarations. For example, a Human Resources package might contain hiring and firing procedures. Once written, your general-purpose package is compiled, then stored in an Oracle database, where its contents can be shared by many applications.

This chapter discusses the following topics:

- What Is a PL/SQL Package?
- Advantages of PL/SQL Packages
- Understanding The Package Spec
- Understanding The Package Body
- Some Examples of Package Features
- Private Versus Public Items in Packages
- Overloading Packaged Subprograms
- How Package STANDARD Defines the PL/SQL Environment
- Overview of Product-Specific Packages
What Is a PL/SQL Package?

A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The specification (spec for short) is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the spec.

As Figure 9–1 shows, you can think of the spec as an operational interface and of the body as a "black box." You can debug, enhance, or replace a package body without changing the interface (package spec) to the package.

Figure 9–1 Package Interface

To create packages, use the CREATE PACKAGE statement, which you can execute interactively from SQL*Plus. Here is the syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
  [AUTHID {CURRENT_USER | DEFINER}]
  {IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
  [collection_type_definition ...
  [record_type_definition ...]
  [subtype_definition ...]
  [collection_declaration ...
  [constant_declaration ...
  [exception_declaration ...
  [object_declaration ...
  [record_declaration ...
  [variable_declaration ...]
```
What Is a PL/SQL Package?

The spec holds **public declarations**, which are visible to your application. You must declare subprograms at the end of the spec after all other items (except pragmas that name a specific function; such pragmas must follow the function spec).

The body holds implementation details and **private declarations**, which are hidden from your application. Following the declarative part of the package body is the optional initialization part, which typically holds statements that initialize package variables.

The **AUTHID** clause determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights Versus Definer Rights" on page 8-47.

A **call spec** lets you publish a Java method or external C function in the Oracle data dictionary. The call spec publishes the routine by mapping its name, parameter types, and return type to their SQL counterparts. To learn how to write Java call specs, see *Oracle9i Java Stored Procedures Developer’s Guide*. To learn how to write C call specs, see *Oracle9i Application Developer’s Guide - Fundamentals*. 
Example of a PL/SQL Package

In the example below, you package a record type, a cursor, and two employment procedures. Notice that the procedure hire_employee uses the database sequence empno_seq and the function SYSDATE to insert a new employee number and hire date, respectively.

```sql
CREATE OR REPLACE PACKAGE emp_actions AS  -- spec
    TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
    CURSOR desc_salary RETURN EmpRecTyp;
    PROCEDURE hire_employee (    
        ename  VARCHAR2,
        job    VARCHAR2,
        mgr    NUMBER,
        sal    NUMBER,
        comm   NUMBER,
        deptno NUMBER);
    PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- body
    CURSOR desc_salary RETURN EmpRecTyp IS
        SELECT empno, sal FROM emp ORDER BY sal DESC;
    PROCEDURE hire_employee (    
        ename  VARCHAR2,
        job    VARCHAR2,
        mgr    NUMBER,
        sal    NUMBER,
        comm   NUMBER,
        deptno NUMBER) IS
    BEGIN
        INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,  
            mgr, SYSDATE, sal, comm, deptno);
    END hire_employee;

    PROCEDURE fire_employee (emp_id NUMBER) IS
    BEGIN
        DELETE FROM emp WHERE empno = emp_id;
    END fire_employee;
END emp_actions;

Only the declarations in the package spec are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. So, you can change the body (implementation) without having to recompile calling programs.
Advantages of PL/SQL Packages

Packages offer several advantages: modularity, easier application design, information hiding, added functionality, and better performance.

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. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.

Better Performance
When you call a packaged subprogram for the first time, the whole package is loaded into memory. So, later calls to related subprograms in the package require no disk I/O. Also, packages stop cascading dependencies and thereby avoid unnecessary recompiling. For example, if you change the implementation of a packaged function, Oracle need not recompile the calling subprograms because they do not depend on the package body.
Understanding The Package Spec

The package spec contains public declarations. The scope of these declarations is local to your database schema and global to the package. So, the declared items are accessible from your application and from anywhere in the package. Figure 9–2 illustrates the scoping.

The spec lists the package resources available to applications. All the information your application needs to use the resources is in the spec. For example, the following declaration shows that the function named `fac` takes one argument of type `INTEGER` and returns a value of type `INTEGER`:

```
FUNCTION fac (n INTEGER) RETURN INTEGER;  -- returns n!
```

That is all the information you need to call the function. You need not consider its underlying implementation (whether it is iterative or recursive for example). Only subprograms and cursors have an underlying implementation. So, if a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. Consider the following bodiless package:

```
CREATE PACKAGE trans_data AS  -- bodiless package
    TYPE TimeRec IS RECORD (
        minutes SMALLINT,
        hours   SMALLINT);
    TYPE TransRec IS RECORD (
        category VARCHAR2,
        account  INT,
        amount   REAL,
        time_of  TimeRec);
```

---

9-6  PL/SQL User's Guide and Reference
minimum_balance CONSTANT REAL := 10.00;
number_processed INT;
insufficient_funds EXCEPTION;
END trans_data;

The package trans_data needs no body because types, constants, variables, and exceptions do not have an underlying implementation. Such packages let you define global variables—usable by subprograms and database triggers—that persist throughout a session.

Referencing Package Contents

To reference the types, items, subprograms, and call specs declared within a package spec, use dot notation, as follows:

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 might call the packaged procedure hire_employee from SQL*Plus, as follows:

SQL> CALL emp_actions.hire_employee('TATE', 'CLERK', ...);

In the example below, you call the same procedure from an anonymous PL/SQL block embedded in a Pro*C program. The actual parameters emp_name and job_title are host variables (that is, variables declared in a host environment).

EXEC SQL EXECUTE
BEGIN
  emp_actions.hire_employee(:emp_name, :job_title, ...);

Restrictions

You cannot reference remote packaged variables directly or indirectly. For example, you cannot call the following procedure remotely because it references a packaged variable in a parameter initialization clause:

CREATE PACKAGE random AS
  seed NUMBER;
  PROCEDURE initialize (starter IN NUMBER := seed, ...);

Also, inside a package, you cannot reference host variables.
Understanding The Package Body

The package body implements the package spec. That is, the package body contains the implementation of every cursor and subprogram declared in the package spec. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specs also appear in the package spec.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. So, except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception, as the following example shows:

```sql
CREATE PACKAGE emp_actions AS
  ...
  PROCEDURE calc_bonus (date_hired emp.hiredate%TYPE, ...);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
  ...
  PROCEDURE calc_bonus (date_hired DATE, ...) IS
    -- parameter declaration raises an exception because 'DATE'
    -- does not match 'emp.hiredate%TYPE' word for word
    BEGIN ... END;
END emp_actions;
```

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 called 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 spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. However, the body can still be used to initialize items declared in the package spec.
Some Examples of Package Features

Consider the package below named `emp_actions`. The package spec declares the following types, items, and subprograms:

- **Types** `EmpRecTyp` and `DeptRecTyp`
- **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, call its subprograms, use its cursor, and raise its exception. When you create the package, it is stored in an Oracle database for general use.

```sql
CREATE PACKAGE emp_actions AS
    /* Declare externally visible types, cursor, exception. */
    TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
    TYPE DeptRecTyp IS RECORD (dept_id INT, location VARCHAR2);
    CURSOR desc_salary RETURN EmpRecTyp;
    invalid_salary EXCEPTION;

    /* Declare externally callable subprograms. */
    FUNCTION hire_employee (ename VARCHAR2, job VARCHAR2, mgr REAL, sal REAL, comm REAL, deptno REAL) RETURN INT;
    PROCEDURE fire_employee (emp_id INT);
    PROCEDURE raise_salary (emp_id INT, grade INT, amount REAL);
    FUNCTION nth_highest_salary (n INT) RETURN EmpRecTyp;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
    number_hired INT;  -- visible only in this package

    /* Fully define cursor specified in package. */
    CURSOR desc_salary RETURN EmpRecTyp IS
        SELECT empno, sal FROM emp ORDER BY sal DESC;
```
```plsql
/* Fully define subprograms specified in package. */
FUNCTION hire_employee (ename VARCHAR2, 
job VARCHAR2, 
mgr REAL, 
sal REAL, 
comm REAL, 
depth REAL) RETURN INT IS
  new_empno INT;
BEGIN
  SELECT empno_seq.NEXTVAL INTO new_empno FROM dual;
  INSERT INTO emp VALUES (new_empno, ename, job, 
mgr, SYSDATE, sal, comm, deptno);
  number_hired := number_hired + 1;
  RETURN new_empno;
END hire_employee;

PROCEDURE fire_employee (emp_id INT) IS
BEGIN
  DELETE FROM emp WHERE empno = emp_id;
END fire_employee;

/* Define local function, available only inside package. */
FUNCTION sal_ok (rank INT, salary REAL) RETURN BOOLEAN IS
  min_sal REAL;
  max_sal REAL;
BEGIN
  SELECT losal, hisal INTO min_sal, max_sal FROM salgrade 
WHERE grade = rank;
  RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;

PROCEDURE raise_salary (emp_id INT, grade INT, amount REAL) IS
  salary REAL;
BEGIN
  SELECT sal INTO salary FROM emp WHERE empno = emp_id;
  IF sal_ok(grade, salary + amount) THEN
    UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
  ELSE
    RAISE invalid_salary;
  END IF;
END raise_salary;
```
FUNCTION nth_highest_salary (n INT) 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_ACTIONS');
  number_hired := 0;
END emp_actions;

Remember, the initialization part of a package is run just once, the first time you reference the package. So, in the last example, only one row is inserted into the database table emp_audit. Likewise, the variable number_hired is initialized only once.

Every time the procedure hire_employee is called, 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.

In the next example, you package some typical bank transactions. Assume that debit and credit transactions are entered after business hours through automatic teller machines, then applied to accounts the next morning.

CREATE PACKAGE bank_transactions AS
  /* Declare externally visible constant. */
  minimum_balance CONSTANT REAL := 100.00;
  /* Declare externally callable procedures. */
  PROCEDURE apply_transactions;
  PROCEDURE enter_transaction (acct INT, kind CHAR, amount REAL);
CREATE PACKAGE BODY bank_transactions AS

/* Declare global variable to hold transaction status. */
new_status VARCHAR2(70) := ‘Unknown’;

/* Use forward declarations because apply_transactions
   calls credit_account and debit_account, which are not
   yet declared when the calls are made. */
PROCEDURE credit_account (acct INT, credit REAL);
PROCEDURE debit_account (acct INT, debit REAL);

/* Fully define procedures specified in package. */
PROCEDURE apply_transactions IS

/* Apply pending transactions in transactions table
   to accounts table. Use cursor to fetch rows. */
CURSOR trans_cursor IS
  SELECT acct_id, kind, amount FROM transactions
  WHERE status = ‘Pending’
  ORDER BY time_tag
  FOR UPDATE OF status; -- to lock rows
BEGIN
  FOR trans IN trans_cursor LOOP
    IF trans.kind = ‘D’ THEN
      debit_account(trans.acct_id, trans.amount);
    ELSIF trans.kind = ‘C’ THEN
      credit_account(trans.acct_id, trans.amount);
    ELSE
      new_status := ‘Rejected’;
    END IF;
    UPDATE transactions SET status = new_status
    WHERE CURRENT OF trans_cursor;
  END LOOP;
END apply_transactions;

PROCEDURE enter_transaction (
/* Add a transaction to transactions table. */
acct   INT,
kind   CHAR,
amount REAL) IS
BEGIN
  INSERT INTO transactions
    VALUES (acct, kind, amount, ’Pending’, SYSDATE);
END enter_transaction;
Some Examples of Package Features

/* Define local procedures, available only in package. */

PROCEDURE do_journal_entry (acct INT, kind CHAR, new_bal REAL) IS
BEGIN
    INSERT INTO journal
        VALUES (acct, kind, new_bal, sysdate);
    IF kind = 'D' THEN
        new_status := 'Debit applied';
    ELSE
        new_status := 'Credit applied';
    END IF;
END do_journal_entry;

PROCEDURE credit_account (acct INT, credit REAL) IS
BEGIN
    SELECT balance INTO old_balance FROM accounts
        WHERE acct_id = acct
            FOR UPDATE OF balance;  -- to lock the row
    new_balance := old_balance + credit;
    UPDATE accounts SET balance = new_balance
        WHERE acct_id = acct;
    do_journal_entry(acct, 'C', new_balance);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        new_status := 'Bad account number';
    WHEN OTHERS THEN
        new_status := SUBSTR(SQLERRM,1,70);
END credit_account;

PROCEDURE debit_account (acct INT, debit REAL) IS
BEGIN
    SELECT balance INTO old_balance FROM accounts
        WHERE acct_id = acct
            FOR UPDATE OF balance;  -- to lock the row
    new_balance := old_balance - debit;
    UPDATE accounts SET balance = new_balance
        WHERE acct_id = acct;
    do_journal_entry(acct, 'D', new_balance);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        new_status := 'Bad account number';
    WHEN OTHERS THEN
        insufficient_funds := SUBSTR(SQLERRM,1,70);
END debit_account;
new_balance := old_balance - debit;
IF new_balance >= minimum_balance THEN
    UPDATE accounts SET balance = new_balance
    WHERE acct_id = acct;
    do_journal_entry(acct, ’D’, new_balance);
ELSE
    RAISE insufficient_funds;
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
    new_status := ’Bad account number’;
WHEN insufficient_funds THEN
    new_status := ’Insufficient funds’;
WHEN OTHERS THEN
    new_status := SUBSTR(SQLERRM,1,70);
END debit_account;
END bank_transactions;

In this package, the initialization part is not used.

Private Versus Public Items in Packages

Look again at the package emp_actions. The package body declares a variable named number_hired, which is initialized to zero. Unlike items declared in the spec of emp_actions, items declared in the body are restricted to use within the package. Therefore, PL/SQL code outside the package cannot reference the variable number_hired. Such items are called private.

However, items declared in the spec of emp_actions, such as the exception invalid_salary, are visible outside the package. Therefore, any PL/SQL code can reference the exception invalid_salary. Such items are called public.

When you must 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.

If you must also make the items public, place them in the package spec. For example, the constant minimum_balance declared in the spec of the package bank_transactions is available for general use.
Overloading Packaged Subprograms

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when you want a subprogram to accept similar sets of parameters that have different datatypes. For example, the following package defines two procedures named `journalize`:

CREATE PACKAGE journal_entries AS
...
PROCEDURE journalize (amount REAL, trans_date VARCHAR2);
PROCEDURE journalize (amount REAL, trans_date INT);
END journal_entries;

CREATE PACKAGE BODY journal_entries AS
...
PROCEDURE journalize (amount REAL, trans_date VARCHAR2) IS
BEGIN
    INSERT INTO journal
    VALUES (amount, TO_DATE(trans_date, 'DD-MON-YYYY'));
END journalize;

PROCEDURE journalize (amount REAL, trans_date INT) IS
BEGIN
    INSERT INTO journal
    VALUES (amount, TO_DATE(trans_date, 'J'));
END journalize;
END journal_entries;

The first procedure accepts `trans_date` as a character string, while the second procedure accepts it as a number (the Julian day). Each procedure handles the data appropriately. For the rules that apply to overloaded subprograms, see "Overloading Subprogram Names" on page 8-23.

How Package STANDARD Defines the PL/SQL Environment

A package named `STANDARD` defines the PL/SQL environment. The package spec globally declares types, exceptions, and subprograms, which are available automatically to PL/SQL programs. For example, package `STANDARD` declares function `ABS`, which returns the absolute value of its argument, as follows:

FUNCTION ABS (n NUMBER) RETURN NUMBER;

The contents of package `STANDARD` are directly visible to applications. You do not need to qualify references to its contents by prefixing the package name. For
example, you might call ABS from a database trigger, stored subprogram, Oracle tool, or 3GL application, as follows:

```plsql
abs_diff := ABS(x - y);
```

If you redeclare ABS in a PL/SQL program, your local declaration overrides the global declaration. However, you can still call the built-in function by qualifying the reference to ABS, as follows:

```plsql
abs_diff := STANDARD.ABS(x - y);
```

Most built-in functions are overloaded. For example, package STANDARD contains the following declarations:

```plsql
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves a call to TO_CHAR by matching the number and datatypes of the formal and actual parameters.

### Overview of Product-Specific Packages

Oracle and various Oracle tools are supplied with product-specific packages that help you build PL/SQL-based applications. For example, Oracle is supplied with many utility packages, a few of which are highlighted below. For more information, see Oracle9i Supplied PL/SQL Packages and Types Reference.

### About the DBMS_ALERT Package

Package DBMS_ALERT 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.

### About the DBMS_OUTPUT Package

Package DBMS_OUTPUT enables you to display output from PL/SQL blocks and subprograms, which makes it easier to test and debug them. The procedure `put_line` outputs information to a buffer in the SGA. You display the information
by calling the procedure get_line or by setting SERVEROUTPUT ON in SQL*Plus. For example, suppose you create the following stored procedure:

```plsql
CREATE PROCEDURE calc_payroll (payroll OUT NUMBER) AS
    CURSOR c1 IS SELECT sal, comm FROM emp;
BEGIN
    payroll := 0;
    FOR c1rec IN c1 LOOP
        c1rec.comm := NVL(c1rec.comm, 0);
        payroll := payroll + c1rec.sal + c1rec.comm;
    END LOOP;
    /* Display debug info. */
    dbms_output.put_line('Value of payroll: ' || TO_CHAR(payroll));
END;
```

When you issue the following commands, SQL*Plus displays the value assigned by the procedure to parameter `payroll`:

```sql
SQL> SET SERVEROUTPUT ON
SQL> VARIABLE num NUMBER
SQL> CALL calc_payroll(:num);
Value of payroll: 31225
```

**About the DBMS_PIPE Package**

Package **DBMS_PIPE** 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.

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 routines in C that allow external programs to collect information, then send it through pipes to procedures stored in an Oracle database.

**About the UTL_FILE Package**

Package **UTL_FILE** allows your PL/SQL programs to read and write operating system (OS) text files. It provides a restricted version of standard OS stream file I/O, including open, put, get, and close operations.

When you want to read or write a text file, you call the function `fopen`, which returns a file handle for use in subsequent procedure calls. For example, the
Guidelines for Writing Packages

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.

About the UTL_HTTP Package

Package UTL_HTTP allows your PL/SQL programs to make hypertext transfer protocol (HTTP) callouts. It can retrieve data from the Internet or call Oracle Web Server cartridges. The package has two entry points, each of which accepts a URL (uniform resource locator) string, contacts the specified site, and returns the requested data, which is usually in hypertext markup language (HTML) format.

Guidelines for Writing Packages

When writing packages, keep them as general as possible so they can be reused in future applications. Avoid writing packages that duplicate some feature already provided by Oracle.

Package specs reflect the design of your application. So, define them before the package bodies. Place in a spec only the types, items, and subprograms that must be visible to users of the package. That way, other developers cannot misuse the package by basing their code on irrelevant 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 Oracle to recompile dependent procedures. However, changes to a package spec require Oracle to recompile every stored subprogram that references the package.
Object-oriented programming is rapidly gaining acceptance because it can reduce the cost and time required to build complex applications. In PL/SQL, object-oriented programming is based on object types. They provide abstract templates for real-world objects, and so are an ideal modeling tool. They also provide black-box encapsulation like an integrated component that can be plugged into various electronic devices. To plug an object type into your programs, you need to know only what it does, not how it works.

This chapter discusses the following topics:

- The Role of Abstraction
- What Is an Object Type?
- Why Use Object Types?
- Structure of an Object Type
- Components of an Object Type
- Defining Object Types
- Declaring and Initializing Objects
- Accessing Attributes
- Calling Constructors
- Calling Methods
- Sharing Objects
- Manipulating Objects
The Role of Abstraction

An abstraction is a high-level description or model of a real-world entity. Abstractions keep our daily lives manageable. They help us reason about an object, event, or relationship by suppressing irrelevant detail. For example, to drive a car, you need not know how its engine works. A simple interface consisting of a gearshift, steering wheel, accelerator, and brake, lets you use the car effectively. The details of what happens under the hood are not important for day-to-day driving.

Abstractions are central to the discipline of programming. For example, you use procedural abstraction when you suppress the details of a complex algorithm by writing a procedure and passing it parameters. A single procedure call hides the details of your implementation. To try a different implementation, you simply replace the procedure with another having the same name and parameters. Thanks to abstraction, programs that call the procedure need not be modified.

You use data abstraction when you specify the datatype of a variable. The datatype stipulates a set of values and a set of operations appropriate for those values. For instance, a variable of type POSITIVE can hold only positive integers, and can only be added, subtracted, multiplied, and so on. To use the variable, you need not know how PL/SQL stores integers or implements arithmetic operations; you simply accept the programming interface.

Object types are a generalization of the built-in datatypes found in most programming languages. PL/SQL provides a variety of built-in scalar and composite datatypes, each of which is associated with a set of predefined operations. A scalar type (such as CHAR) has no internal components. A composite type (such as RECORD) has internal components that can be manipulated individually. Like the RECORD type, an object type is a composite type. However, its operations are user-defined, not predefined.

Currently, you cannot define object types within PL/SQL. They must be created and stored in an Oracle database, where they can be shared by many programs. A program that uses object types is called a client program. It can declare and manipulate an object without knowing how the object type represents data or implements operations. This lets you write the program and object type separately, and to change the implementation of the object type without affecting the program. Thus, object types support both procedural and data abstraction.
What Is an Object Type?

An **object type** is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called **attributes**. The functions and procedures that characterize the behavior of the object type are called **methods**.

We usually think of an object (such as a person, car, or bank account) as having attributes and behaviors. For example, a baby has the attributes gender, age, and weight, and the behaviors eat, drink, and sleep. Object types let you maintain this perspective when you sit down to write an application.

When you define an object type using the `CREATE TYPE` statement, you create an abstract template for some real-world object. The template specifies only those attributes and behaviors the object will need in the application environment. For example, an employee has many attributes, but usually only a few are needed to fill the requirements of an application (see Figure 10–1).

![Figure 10–1 Form Follows Function](image)

Suppose you must write a program to allocate employee bonuses. Not all employee attributes are needed to solve this problem. So, you design an abstract employee who has the following problem-specific attributes: name, ID number, department, job title, salary, and rank. Then, you identify the operations needed to handle an abstract employee. For example, you need an operation that lets Management change the rank of an employee.
What Is an Object Type?

Next, you define a set of variables (attributes) to represent the data, and a set of subprograms (methods) to perform the operations. Finally, you encapsulate the attributes and methods in an object type.

The data structure formed by the set of attributes is public (visible to client programs). However, well-behaved programs do not manipulate it directly. Instead, they use the set of methods provided. That way, the employee data is kept in a proper state.

At run time, when the data structure is filled with values, you have created an instance of an abstract employee. You can create as many instances (usually called objects) as you need. Each object has the name, number, job title, and so on of an actual employee (see Figure 10–2). This data is accessed or changed only by the methods associated with it. Thus, object types let you create objects with well-defined attributes and behavior.

Figure 10–2  Object Type and Objects (Instances) of that Type
Why Use Object Types?

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. It also allows 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. Object types minimize side effects by allowing access to data only through approved operations. Also, object types hide implementation details, so that you can change the details without affecting client programs.

Object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. Moreover, object types map directly into classes defined in object-oriented languages such as Java and C++. Now your programs can better reflect the world they are trying to simulate.

Structure of an Object Type

Like a package, an object type has two parts: a specification and a body (refer to Figure 10–3). The specification (or spec for short) is the interface to your applications; it declares a data structure (set of attributes) along with the operations (set of methods) needed to manipulate the data. The body fully defines the methods, and so implements the spec.

Figure 10–3  Object Type Structure

<table>
<thead>
<tr>
<th>spec</th>
<th>public interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute declarations</td>
<td></td>
</tr>
<tr>
<td>method specs</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>body</th>
<th>private implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>method bodies</td>
<td></td>
</tr>
</tbody>
</table>

All the information a client program needs to use the methods is in the spec. Think of the spec as an operational interface and of the body as a black box. You can debug, enhance, or replace the body without changing the spec—and without affecting client programs.
In an object type spec, all attributes must be declared before any methods. Only subprograms have an underlying implementation. So, if an object type spec declares only attributes, the object type body is unnecessary. You cannot declare attributes in the body. All declarations in the object type spec are public (visible outside the object type).

To understand the structure better, study the example below, in which an object type for complex numbers is defined. For now, it is enough to know that a complex number has two parts, a real part and an imaginary part, and that several arithmetic operations are defined for complex numbers.

```plsql
CREATE TYPE Complex AS OBJECT (  
rpart REAL,  -- attribute  
ipart REAL,  
MEMBER FUNCTION plus (x Complex) RETURN Complex,  -- method  
MEMBER FUNCTION less (x Complex) RETURN Complex,  
MEMBER FUNCTION times (x Complex) RETURN Complex,  
MEMBER FUNCTION divby (x Complex) RETURN Complex
);

CREATE TYPE BODY Complex AS  
MEMBER FUNCTION plus (x Complex) RETURN Complex IS  
  BEGIN  
  RETURN Complex(rpart + x.rpart, ipart + x.ipart);  
  END plus;

MEMBER FUNCTION less (x Complex) RETURN Complex IS  
  BEGIN  
  RETURN Complex(rpart - x.rpart, ipart - x.ipart);  
  END less;

MEMBER FUNCTION times (x Complex) RETURN Complex IS  
  BEGIN  
  RETURN Complex(rpart * x.rpart - ipart * x.ipart,  
                  rpart * x.ipart + ipart * x.rpart);  
  END times;

MEMBER FUNCTION divby (x Complex) RETURN Complex IS  
  z REAL := x.rpart**2 + x.ipart**2;  
  BEGIN  
  RETURN Complex((rpart * x.rpart + ipart * x.ipart) / z,  
                  (ipart * x.rpart - rpart * x.ipart) / z);  
  END divby;
END;
```
Components of an Object Type

An object type encapsulates data and operations. You can declare attributes and methods in an object type spec, but not constants, exceptions, cursors, or types. You must declare at least one attribute (the maximum is 1000). Methods are optional.

Attributes

Like a variable, an attribute is declared with a name and datatype. The name must be unique within the object type (but can be reused in other object types). The datatype can be any Oracle type except

- LONG and LONG RAW
- NCHAR, NCLOB, and NVARCHAR2
- ROWID and UROWID
- The PL/SQL-specific types BINARY_INTEGER (and its subtypes), BOOLEAN, PLS_INTEGER, RECORD, REF_CURSOR, %TYPE, and %ROWTYPE
- Types defined inside a PL/SQL package

You cannot initialize an attribute in its declaration using the assignment operator or DEFAULT clause. Also, you cannot impose the NOT NULL constraint on an attribute. However, objects can be stored in database tables on which you can impose constraints.

The kind of data structure formed by a set of attributes depends on the real-world object being modeled. For example, to represent a rational number, which has a numerator and a denominator, you need only two INTEGER variables. On the other hand, to represent a college student, you need several VARCHAR2 variables to hold a name, address, phone number, status, and so on, plus a VARRAY variable to hold courses and grades.

The data structure can be very complex. For example, the datatype of an attribute can be another object type (called a nested object type). That lets you build a complex object type from simpler object types. Some object types such as queues, lists, and trees are dynamic, meaning that they can grow as they are used. Recursive object types, which contain direct or indirect references to themselves, allow for highly sophisticated data models.
Methods

In general, a method is a subprogram declared in an object type spec using the keyword MEMBER or STATIC. The method cannot have the same name as the object type or any of its attributes. MEMBER methods are invoked on instances, as in

`instance_expression.method()`

However, STATIC methods are invoked on the object type, not its instances, as in

`object_type_name.method()`

Like packaged subprograms, methods have two parts: a specification and a body. The specification (spec for short) consists of a method name, an optional parameter list, and, for functions, a return type. The body is the code that executes to perform a specific task.

For each method spec in an object type spec, there must either be a corresponding method body in the object type body, or the method must be declared NOT INSTANTIABLE to indicate that the body is only present in subtypes of this type. To match method specs and bodies, the PL/SQL compiler does a token-by-token comparison of their headers. The headers must match exactly.

Like an attribute, a formal parameter is declared with a name and datatype. However, the datatype of a parameter cannot be size-constrained. The datatype can be any Oracle type except those disallowed for attributes. (See "Attributes" on page 10-7.) The same restrictions apply to return types.

Allowed Languages for Methods

Oracle lets you implement object methods in PL/SQL, Java or C. You can implement type methods in Java or C by providing a call specification in your type. A call spec publishes a Java method or external C function in the Oracle data dictionary. It publishes the routine by mapping its name, parameter types, and return type to their SQL counterparts. To learn how to write Java call specs, see Oracle9i Java Stored Procedures Developer’s Guide. To learn how to write C call specs, see Oracle9i Application Developer’s Guide - Fundamentals.

Parameter SELF

MEMBER methods accept a built-in parameter named SELF, which is an instance of the object type. Whether declared implicitly or explicitly, it is always the first parameter passed to a MEMBER method. However, STATIC methods cannot accept or reference SELF.
Components of an Object Type

In the method body, SELF denotes the object whose method was invoked. For example, method `transform` declares SELF as an IN OUT parameter:

```
CREATE TYPE Complex AS OBJECT (  
    MEMBER FUNCTION transform (SELF IN OUT Complex) ...  
)
```

You cannot specify a different datatype for SELF. In MEMBER functions, if SELF is not declared, its parameter mode defaults to IN. However, in MEMBER procedures, if SELF is not declared, its parameter mode defaults to IN OUT. You cannot specify the OUT parameter mode for SELF.

As the following example shows, methods can reference the attributes of SELF without a qualifier:

```
CREATE TYPE Rational AS OBJECT (  
    num INTEGER, 
    den INTEGER, 
    MEMBER PROCEDURE normalize, 
    ...  
);

CREATE TYPE BODY Rational AS  
    MEMBER PROCEDURE normalize IS  
    g INTEGER;  
BEGIN  
    g := gcd(SELF.num, SELF.den);  
    g := gcd(num, den);  -- equivalent to previous statement  
    num := num / g;  
    den := den / g;  
    END normalize;  
    ...  
END;
```
From a SQL statement, if you call a MEMBER method on a null instance (that is, SELF is null), the method is not invoked and a null is returned. From a procedural statement, if you call a MEMBER method on a null instance, PL/SQL raises the predefined exception SELF_IS_NULL before the method is invoked.

**Overloading**

Like packaged subprograms, methods of the same kind (functions or procedures) can be overloaded. That is, you can use the same name for different methods if their formal parameters differ in number, order, or datatype family. When you call one of the methods, PL/SQL finds it by comparing the list of actual parameters with each list of formal parameters.

A subtype can also overload methods it inherits from its supertype. In this case, the methods can have exactly the same formal parameters.

You cannot overload two methods if their formal parameters differ only in parameter mode. Also, you cannot overload two member functions that differ only in return type. For more information, see "Overloading Subprogram Names" on page 8-23.

**MAP and ORDER Methods**

The values of a scalar datatype such as CHAR or REAL have a predefined order, which allows them to be compared. But instances of an object type have no predefined order. To put them in order for comparison or sorting purposes, PL/SQL calls a MAP method supplied by you. In the following example, the keyword MAP indicates that method convert() orders Rational objects by mapping them to REAL values:

```sql
CREATE TYPE Rational AS OBJECT (
    num INTEGER,
    den INTEGER,
    MAP MEMBER FUNCTION convert RETURN REAL,
    ...
);

CREATE TYPE BODY Rational AS
    MAP MEMBER FUNCTION convert RETURN REAL IS
    BEGIN
        RETURN num / den;
    END convert;
    ...
END;
```

```sql
```
PL/SQL uses the ordering to evaluate Boolean expressions such as $x > y$, and to do comparisons implied by the DISTINCT, GROUP BY, and ORDER BY clauses. MAP method convert() returns the relative position of an object in the ordering of all Rational objects.

An object type can contain only one MAP method. It accepts the built-in parameter SELF and returns one of the following scalar types: DATE, NUMBER, VARCHAR2, or an ANSI SQL type such as CHARACTER or REAL.

Alternatively, you can supply PL/SQL with an ORDER method. An object type can contain only one ORDER method, which must be a function that returns a numeric result. In the following example, the keyword ORDER indicates that method match() compares two objects:

```sql
CREATE TYPE Customer AS OBJECT (  
  id   NUMBER,  
  name VARCHAR2(20),  
  addr VARCHAR2(30),  
  ORDER MEMBER FUNCTION match (c Customer) RETURN INTEGER  
);  

CREATE TYPE BODY Customer AS  
ORDER MEMBER FUNCTION match (c Customer) RETURN INTEGER IS  
BEGIN  
  IF id < c.id THEN  
    RETURN -1;  -- any negative number will do  
  ELSIF id > c.id THEN  
    RETURN 1;  -- any positive number will do  
  ELSE  
    RETURN 0;  
  END IF;  
END;  
END;  
```

Every ORDER method takes just two parameters: the built-in parameter SELF and another object of the same type. If c1 and c2 are Customer objects, a comparison such as c1 > c2 calls method match automatically. The method returns a negative number, zero, or a positive number signifying that SELF is respectively less than, equal to, or greater than the other parameter. If either parameter passed to an ORDER method is null, the method returns a null.

**Guidelines** A MAP method, acting like a hash function, maps object values into scalar values, which are then compared using operators such as $<$, $=$, and so on. An ORDER method simply compares one object value to another.
You can declare a `MAP` method or an `ORDER` method but not both. If you declare either method, you can compare objects in SQL and procedural statements. However, if you declare neither method, you can compare objects only in SQL statements and only for equality or inequality. (Two objects of the same type are equal only if the values of their corresponding attributes are equal.)

When sorting or merging a large number of objects, use a `MAP` method. One call maps all the objects into scalars, then sorts the scalars. An `ORDER` method is less efficient because it must be called repeatedly (it can compare only two objects at a time). You must use a `MAP` method for hash joins because PL/SQL hashes on the object value.

**Constructor Methods**

Every object type has a constructor method (constructor for short), which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type.

Oracle generates a default constructor for every object type. The formal parameters of the constructor match the attributes of the object type. That is, the parameters and attributes are declared in the same order and have the same names and datatypes. PL/SQL never calls a constructor implicitly, so you must call it explicitly. For more information, see "Calling Constructors" on page 10-28.

**Changing Attributes and Methods of an Existing Object Type (Type Evolution)**

You can use the `ALTER TYPE` statement to add, modify, or drop attributes, and add or drop methods of an existing object type:

```sql
CREATE TYPE Person_typ AS OBJECT
     ( name CHAR(20),
       ssn CHAR(12),
       address VARCHAR2(100));
CREATE TYPE Person_nt IS TABLE OF Person_typ;
CREATE TYPE dept_typ AS OBJECT
     ( mgr Person_typ,
       emps Person_nt);
CREATE TABLE dept OF dept_typ;

-- Add new attributes to Person_typ and propagate the change
to Person_nt and dept_typ
ALTER TYPE Person_typ ADD ATTRIBUTE (picture BLOB, dob DATE)
CASCADE NOT INCLUDING TABLE DATA;

CREATE TYPE mytype AS OBJECT (attr1 NUMBER, attr2 NUMBER);
```
When a procedure is compiled, it always uses the current version of any object types it references. Existing procedures on the server that reference an object type are invalidated when the type is altered, and are automatically recompiled the next time the procedure is called. You must manually recompile any procedures on the client side that reference types that are altered.

If you drop a method from a supertype, you might have to make changes to subtypes that override that method. You can find if any subtypes are affected by using the CASCADE option of ALTER TYPE; the statement is rolled back if any subtypes override the method. To successfully drop the method from the supertype, you can:

- Drop the method permanently from the subtype first.
- Drop the method in the subtype, then add it back later using ALTER TYPE without the OVERRIDING keyword.

For more information about the ALTER TYPE statement, see Oracle9i SQL Reference. For guidelines about using type evolution in your applications, and options for changing other types and data that rely on see Oracle9i Application Developer's Guide - Object-Relational Features.
Defining Object Types

[INSTANTIABLE| NOT INSTANTIABLE] MEMBER function_spec,
[MEMBER | STATIC] {subprogram_spec | call_spec}
[, {MEMBER | STATIC} {subprogram_spec | call_spec}] ...
) [{FINAL| NOT FINAL}] [ {INSTANTIABLE| NOT INSTANTIABLE}];

[CREATE [OR REPLACE] TYPE BODY type_name {IS | AS}
{ {MAP | ORDER} MEMBER function_body;
  | {MEMBER | STATIC} {subprogram_body | call_spec};

[MEMBER | STATIC} {subprogram_body | call_spec};...
END;}]

The AUTHID clause determines whether all member methods execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights Versus Definer Rights" on page 8-47.

Overview of PL/SQL Type Inheritance

PL/SQL supports a single-inheritance model. You can define subtypes of object types. These subtypes contain all the attributes and methods of the parent type (or supertype). The subtypes can also contain additional attributes and additional methods, and can override methods from the supertype.

You can define whether or not subtypes can be derived from a particular type. You can also define types and methods that cannot be instantiated directly, only by declaring subtypes that instantiate them.

Some of the type properties can be changed dynamically with the ALTER TYPE statement. When changes are made to the supertype, either through ALTER TYPE or by redefining the supertype, the subtypes automatically reflect those changes.

You can use the TREAT operator in queries to return only those objects that are of a specified subtype. You can use the IS OF predicate to test whether an object is a member of one or more specified types or their subtypes.

The values from the REF and DEREF functions can represent either the declared type of the table or view, or one or more of its subtypes.

See the Oracle9i Application Developer's Guide - Object-Relational Features for more detail on all these object-relational features.

Examples of PL/SQL Type Inheritance

-- Create a supertype from which several subtypes will be derived.
CREATE TYPE Person_typ AS OBJECT ( ssn NUMBER, name VARCHAR2(30),
address VARCHAR2(100)) NOT FINAL;

-- Derive a subtype that has all the attributes of the supertype, 
-- plus some additional attributes.
CREATE TYPE Student_typ UNDER Person_typ ( deptid NUMBER, major VARCHAR2(30)) NOT FINAL;

-- Because Student_typ is declared NOT FINAL, you can derive 
-- further subtypes from it.
CREATE TYPE PartTimeStudent_typ UNDER Student_typ( numhours NUMBER);

-- Derive another subtype. Because it has the default attribute 
-- FINAL, you cannot use Employee_typ as a supertype and derive 
-- subtypes from it.
CREATE TYPE Employee_typ UNDER Person_typ( empid NUMBER, mgr VARCHAR2(30));

-- Define an object type that can be a supertype. Because the 
-- member function is FINAL, it cannot be overridden in any 
-- subtypes.
CREATE TYPE T AS OBJECT (...) MEMBER PROCEDURE Print(), FINAL MEMBER FUNCTION foo(x NUMBER)...) NOT FINAL;

-- We never want to create an object of this supertype. By using 
-- NOT INSTANTIABLE, we force all objects to use one of the subtypes 
-- instead, with specific implementations for the member functions.
CREATE TYPE Address_typ AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;

-- These subtypes can provide their own implementations of 
-- member functions, such as for validating phone numbers and 
-- postal codes. Because there is no "generic" way of doing these 
-- things, only objects of these subtypes can be instantiated.
CREATE TYPE USAddress_typ UNDER Address_typ(...);
CREATE TYPE IntlAddress_typ UNDER Address_typ(...);

-- Return REFs for those Person_typ objects that are instances of 
-- the Student_typ subtype, and NULL REFs otherwise.
SELECT TREAT(REF(p) AS REF Student_typ) FROM Person_v p;

-- Example of using TREAT for assignment...

-- Return REFs for those Person_typ objects that are instances of 
-- Employee_type or Student_typ, or any of their subtypes.
SELECT REF(p) FROM Person_v p WHERE VALUE(p) IS OF (Employee_typ, Student_typ);

-- Similar to above, but do not allow any subtypes of Student_typ.
SELECT REF(p) FROM Person_v p WHERE VALUE(p) IS OF(ONLY Student_typ);

-- The results of REF and DEREF can include objects of Person_typ
-- and its subtypes such as Employee_typ and Student_typ.
SELECT REF(p) FROM Person_v p;
SELECT DEREF(REF(p)) FROM Person_v p;

Object Type Example: Stack

A stack holds an ordered collection of data items. Stacks have a top and a bottom. Items can be added or removed only at the top. So, the last item added to a stack is the first item removed. (Think of the stack of clean serving trays in a cafeteria.) The operations push and pop update the stack while preserving last in, first out (LIFO) behavior.

Stacks have many applications. For example, they are used in systems programming to prioritize interrupts and to manage recursion. The simplest implementation of a stack uses an integer array, with one end of the array representing the top of the stack.

PL/SQL provides the datatype VARRAY, which lets you declare variable-size arrays (varrays for short). To declare a varray attribute, you must first define its type. However, you cannot define types in an object type spec. So, you define a standalone varray type, specifying its maximum size, as follows:

CREATE TYPE IntArray AS VARRAY(25) OF INTEGER;

Now, you can write the object type spec:

CREATE TYPE Stack AS OBJECT (  max_size INTEGER,
    top      INTEGER,
    position IntArray,
    MEMBER PROCEDURE initialize,
    MEMBER FUNCTION full RETURN BOOLEAN,
    MEMBER FUNCTION empty RETURN BOOLEAN,
    MEMBER PROCEDURE push (n IN INTEGER),
    MEMBER PROCEDURE pop (n OUT INTEGER) );
Finally, you write the object type body:

```
CREATE TYPE BODY Stack AS
    MEMBER PROCEDURE initialize IS
        BEGIN
            top := 0;
            /* Call constructor for varray and set element 1 to NULL. */
            position := IntArray(NULL);
            max_size := position.LIMIT;  -- get varray size constraint
            position.EXTEND(max_size - 1, 1);  -- copy element 1 into 2..25
        END initialize;

    MEMBER FUNCTION full RETURN BOOLEAN IS
        BEGIN
            RETURN (top = max_size);  -- return TRUE if stack is full
        END full;

    MEMBER FUNCTION empty RETURN BOOLEAN IS
        BEGIN
            RETURN (top = 0);  -- return TRUE if stack is empty
        END empty;

    MEMBER PROCEDURE push (n IN INTEGER) IS
        BEGIN
            IF NOT full THEN
                top := top + 1;  -- push integer onto stack
                position(top) := n;
            ELSE  -- stack is full
                RAISE_APPLICATION_ERROR(-20101, 'stack overflow');
            END IF;
        END push;

    MEMBER PROCEDURE pop (n OUT INTEGER) IS
        BEGIN
            IF NOT empty THEN
                n := position(top);
                top := top - 1;  -- pop integer off stack
            ELSE  -- stack is empty
                RAISE_APPLICATION_ERROR(-20102, 'stack underflow');
            END IF;
        END pop;
    END;
```
In member procedures push and pop, you use the built-in procedure raise_application_error to issue user-defined error messages. That way, you report errors to the client program and avoid returning unhandled exceptions to the host environment. The client program gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS exception handler. In the following example, when an exception is raised, you print the corresponding Oracle error message:

```sql
DECLARE
    ...
BEGIN
    ... 
EXCEPTION
    WHEN OTHERS THEN
        dbms_output.put_line(SQLERRM);
END;
```

Alternatively, the program can use pragma EXCEPTION_INIT to map the error numbers returned by raise_application_error to named exceptions, as the following example shows:

```sql
DECLARE
    stack_overflow  EXCEPTION;
    stack_underflow EXCEPTION;
    PRAGMA EXCEPTION_INIT(stack_overflow, -20101);
    PRAGMA EXCEPTION_INIT(stack_underflow, -20102);
BEGIN
    ...
EXCEPTION
    WHEN stack_overflow THEN
        ...
END;
```

**Object Type Example: Ticket_Booth**

Consider a chain of movie theaters, each theater with three screens. Each theater has a ticket booth where tickets for three different movies are sold. All tickets are priced at $3.00. Periodically, ticket receipts are collected and the stock of tickets is replenished.

Before defining an object type that represents a ticket booth, you must consider the data and operations needed. For a simple ticket booth, the object type needs attributes for the ticket price, quantity of tickets on hand, and receipts. It also needs
methods for the following operations: purchase ticket, take inventory, replenish 
stock, and collect receipts.

For receipts, you use a three-element varray. Elements 1, 2, and 3 record the ticket 
receipts for movies 1, 2, and 3, respectively. To declare a varray attribute, you must 
first define its type, as follows:

```
CREATE TYPE RealArray AS VARRAY(3) OF REAL;
```

Now, you can write the object type spec:

```
CREATE TYPE Ticket_Booth AS OBJECT (  
    price REAL,  
    qty_on_hand INTEGER,  
    receipts RealArray,  
    MEMBER PROCEDURE initialize,  
    MEMBER PROCEDURE purchase (    
        movie INTEGER,    
        amount REAL,    
        change OUT REAL),  
    MEMBER FUNCTION inventory RETURN INTEGER,  
    MEMBER PROCEDURE replenish (quantity INTEGER),  
    MEMBER PROCEDURE collect (movie INTEGER, amount OUT REAL)  
);
```

Finally, you write the object type body:

```
CREATE TYPE BODY Ticket_Booth AS  
    MEMBER PROCEDURE initialize IS  
        BEGIN  
            price := 3.00;  
            qty_on_hand := 5000;  -- provide initial stock of tickets  
            -- call constructor for varray and set elements 1..3 to zero  
            receipts := RealArray(0,0,0);  
        END initialize;

    MEMBER PROCEDURE purchase (    
        movie INTEGER,    
        amount REAL,    
        change OUT REAL) IS  
        BEGIN  
            IF qty_on_hand = 0 THEN  
                RAISE_APPLICATION_ERROR(-20103, 'out of stock');  
            END IF;  
            IF amount >= price THEN  
                qty_on_hand := qty_on_hand - 1;
```
MEMBER FUNCTION inventory RETURN INTEGER IS
BEGIN
  RETURN qty_on_hand;
END inventory;

MEMBER PROCEDURE replenish (quantity INTEGER) IS
BEGIN
  qty_on_hand := qty_on_hand + quantity;
END replenish;

MEMBER PROCEDURE collect (movie INTEGER, amount OUT REAL) IS
BEGIN
  amount := receipts(movie);  -- get receipts for a given movie
  receipts(movie) := 0;  -- reset receipts to zero
END collect;
END;

Object Type Example: Bank_Account

Before defining an object type that represents a bank account, you must consider the data and operations needed. For a simple bank account, the object type needs attributes for an account number, balance, and status. It also needs methods for the following operations: open account, verify account number, close account, deposit money, withdraw money, and return balance.

First, you write the object type spec, as follows:

CREATE TYPE Bank_Account AS OBJECT (  
  acct_number INTEGER(5),  
  balance REAL,  
  status VARCHAR2(10),  
  MEMBER PROCEDURE open (amount IN REAL),  
  MEMBER PROCEDURE verify_acct (num IN INTEGER),  
  MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL),  
  MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL),  
  MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL),  
  MEMBER FUNCTION curr_bal (num IN INTEGER) RETURN REAL  
);
Then, you write the object type body:

```sql
CREATE TYPE BODY Bank_Account AS
    MEMBER PROCEDURE open (amount IN REAL) IS
        -- open account with initial deposit
        BEGIN
            IF NOT amount > 0 THEN
                RAISE_APPLICATION_ERROR(-20104, 'bad amount');
            END IF;
            SELECT acct_sequence.NEXTVAL INTO acct_number FROM dual;
            status := 'open';
            balance := amount;
        END open;

    MEMBER PROCEDURE verify_acct (num IN INTEGER) IS
        -- check for wrong account number or closed account
        BEGIN
            IF (num <> acct_number) THEN
                RAISE_APPLICATION_ERROR(-20105, 'wrong number');
            ELSIF (status = 'closed') THEN
                RAISE_APPLICATION_ERROR(-20106, 'account closed');
            END IF;
        END verify_acct;

    MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL) IS
        -- close account and return balance
        BEGIN
            verify_acct(num);
            status := 'closed';
            amount := balance;
        END close;

    MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL) IS
        BEGIN
            verify_acct(num);
            IF NOT amount > 0 THEN
                RAISE_APPLICATION_ERROR(-20104, 'bad amount');
            END IF;
            balance := balance + amount;
        END deposit;

    MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL) IS
        -- if account has enough funds, withdraw
        BEGIN
            verify_acct(num);
            IF NOT amount > 0 THEN
                RAISE_APPLICATION_ERROR(-20104, 'bad amount');
            END IF;
            balance := balance + amount;
        END withdraw;
```

BEGIN
    verify_acct(num);
    IF amount <= balance THEN
        balance := balance - amount;
    ELSE
        RAISE_APPLICATION_ERROR(-20107, 'insufficient funds');
    END IF;
END withdraw;

MEMBER FUNCTION curr_bal (num IN INTEGER)
    RETURN REAL IS
BEGIN
    verify_acct(num);
    RETURN balance;
END curr_bal;
END;

Object Type Example: Rational Numbers

A rational number is a number expressible as the quotient of two integers, a numerator and a denominator. Like most languages, PL/SQL does not have a rational number type or predefined operations on rational numbers. Let us remedy that omission by defining object type Rational. First, you write the object type spec, as follows:

CREATE TYPE Rational AS OBJECT (
    num INTEGER,
    den INTEGER,
    MAP MEMBER FUNCTION convert RETURN REAL,
    MEMBER PROCEDURE normalize,
    MEMBER FUNCTION reciprocal RETURN Rational,
    MEMBER FUNCTION plus (x Rational) RETURN Rational,
    MEMBER FUNCTION less (x Rational) RETURN Rational,
    MEMBER FUNCTION times (x Rational) RETURN Rational,
    MEMBER FUNCTION divby (x Rational) RETURN Rational,
    PRAGMA RESTRICT_REFERENCES (DEFAULT, RNDS, WNDNS, RNPS, WNPS)
);

PL/SQL does not allow the overloading of operators. So, you must define methods named plus(), less() (the word minus is reserved), times(), and divby() instead of overloading the infix operators +, -, *, and /.

Next, you create the following standalone stored function, which will be called by method normalize():
CREATE FUNCTION gcd (x INTEGER, y INTEGER) RETURN INTEGER AS
  -- find greatest common divisor of x and y
  ans INTEGER;
BEGIN
  IF (y <= x) AND (x MOD y = 0) THEN
    ans := y;
  ELSIF x < y THEN
    ans := gcd(y, x);  -- recursive call
  ELSE
    ans := gcd(y, x MOD y);  -- recursive call
  END IF;
  RETURN ans;
END;

Then, you write the object type body, as follows:

CREATE TYPE BODY Rational AS
  MAP MEMBER FUNCTION convert RETURN REAL IS
    -- convert rational number to real number
    BEGIN
      RETURN num / den;
    END convert;

  MEMBER PROCEDURE normalize IS
    -- reduce fraction num / den to lowest terms
    g INTEGER;
    BEGIN
      g := gcd(num, den);
      num := num / g;
      den := den / g;
    END normalize;

  MEMBER FUNCTION reciprocal RETURN Rational IS
    -- return reciprocal of num / den
    BEGIN
      RETURN Rational(den, num);  -- call constructor
    END reciprocal;

  MEMBER FUNCTION plus (x Rational) RETURN Rational IS
    -- return sum of SELF + x
    r Rational;
    BEGIN
      r := Rational(num * x.den + x.num * den, den * x.den);
      r.normalize;
      RETURN r;
    END;
Declaring and Initializing Objects

Once an object type is defined and installed in the schema, you can use it to declare objects in any PL/SQL block, subprogram, or package. For example, you can use the object type to specify the datatype of an attribute, column, variable, bind variable, record field, table element, formal parameter, or function result. At runtime, instances of the object type are created; that is, objects of that type are instantiated. Each object can hold different values.

Such objects follow the usual scope and instantiation rules. In a block or subprogram, local objects are instantiated when you enter the block or subprogram and cease to exist when you exit. In a package, objects are instantiated when you first reference the package and cease to exist when you end the database session.
Declaring Objects

You can use object types wherever built-in types such as CHAR or NUMBER can be used. In the block below, you declare object r of type Rational. Then, you call the constructor for object type Rational to initialize the object. The call assigns the values 6 and 8 to attributes num and den, respectively.

```sql
DECLARE
  r Rational;
BEGIN
  r := Rational(6, 8);
  dbms_output.put_line(r.num);  -- prints 6
END;
```

You can declare objects as the formal parameters of functions and procedures. That way, you can pass objects to stored subprograms and from one subprogram to another. In the next example, you use object type Account to specify the datatype of a formal parameter:

```sql
DECLARE
  ...
  PROCEDURE open_acct (new_acct IN OUT Account) IS ...
END;
```

In the following example, you use object type Account to specify the return type of a function:

```sql
DECLARE
  ...
  FUNCTION get_acct (acct_id IN INTEGER) RETURN Account IS ...
END;
```

Initializing Objects

Until you initialize an object by calling the constructor for its object type, the object is *atomically null*. That is, the object itself is null, not just its attributes. Consider the following example:

```sql
DECLARE
  r Rational;  -- r becomes atomically null
BEGIN
  r := Rational(2,3);  -- r becomes 2/3
END;
```

A null object is never equal to another object. In fact, comparing a null object with any other object always yields NULL. Also, if you assign an atomically null object to another object, the other object becomes atomically null (and must be reinitialized). Likewise, if you assign the non-value NULL to an object, the object becomes atomically null, as the following example shows:
Declaring and Initializing Objects

DECLARE
    r Rational;
BEGIN
    r Rational := Rational(1,2);  -- r becomes 1/2
    r := NULL;  -- r becomes atomically null
    IF r IS NULL THEN ...  -- condition yields TRUE

A good programming practice is to initialize an object in its declaration, as shown in the following example:

DECLARE
    r Rational := Rational(2,3);  -- r becomes 2/3

How PL/SQL Treats Uninitialized Objects

In an expression, attributes of an uninitialized object evaluate to NULL. Trying to assign values to attributes of an uninitialized object raises the predefined exception ACCESS_INTO_NULL. When applied to an uninitialized object or its attributes, the IS NULL comparison operator yields TRUE.

The following example illustrates the difference between null objects and objects with null attributes:

DECLARE
    r Rational;  -- r is atomically null
BEGIN
    IF r IS NULL THEN ...  -- yields TRUE
    IF r.num IS NULL THEN ...  -- yields TRUE
    r := Rational(NULL, NULL);  -- initializes r
    r.num := 4;  -- succeeds because r is no longer atomically null
    -- even though all its attributes are null
    r := NULL;  -- r becomes atomically null again
    r.num := 4;  -- raises ACCESS_INTO_NULL
EXCEPTION
    WHEN ACCESS_INTO_NULL THEN
        ...
END;

Calls to methods of an uninitialized object raise the predefined exception NULL_SELF_DISPATCH. When passed as arguments to IN parameters, attributes of an uninitialized object evaluate to NULL. When passed as arguments to OUT or IN OUT parameters, they raise an exception if you try to write to them.
Accessing Attributes

You can refer to an attribute only by name (not by its position in the object type). To access or change the value of an attribute, you use dot notation. In the example below, you assign the value of attribute \texttt{den} to variable \texttt{denominator}. Then, you assign the value stored in variable \texttt{numerator} to attribute \texttt{num}.

\begin{verbatim}
DECLARE
  r Rational := Rational(NULL, NULL);
  numerator INTEGER;
  denominator INTEGER;
BEGIN
  ... 
  denominator := r.den;
  r.num := numerator;
END;
\end{verbatim}

Attribute names can be chained, which lets you access the attributes of a nested object type. For example, suppose you define object types \texttt{Address} and \texttt{Student}, as follows:

\begin{verbatim}
CREATE TYPE Address AS OBJECT (
  street   VARCHAR2(30),
  city     VARCHAR2(20),
  state    CHAR(2),
  zip_code VARCHAR2(5)
);

CREATE TYPE Student AS OBJECT (
  name         VARCHAR2(20),
  home_address Address,
  phone_number VARCHAR2(10),
  status       VARCHAR2(10),
  advisor_name VARCHAR2(20),
  ... 
);
\end{verbatim}

Notice that \texttt{zip_code} is an attribute of object type \texttt{Address} and that \texttt{Address} is the datatype of attribute \texttt{home_address} in object type \texttt{Student}. If \texttt{s} is a \texttt{Student} object, you access the value of its \texttt{zip_code} attribute as follows:

\begin{verbatim}
s.home_address.zip_code
\end{verbatim}
Calling Constructors

Calls to a constructor are allowed wherever function calls are allowed. Like all functions, a constructor is called as part of an expression, as the following example shows:

```
DECLARE
  r1 Rational := Rational(2, 3);
FUNCTION average (x Rational, y Rational) RETURN Rational IS
BEGIN
  ...
END;
BEGIN
  r1 := average(Rational(3, 4), Rational(7, 11));
  IF (Rational(5, 8) > r1) THEN
    ...
  END IF;
END;
```

When you pass parameters to a constructor, the call assigns initial values to the attributes of the object being instantiated. You must supply a parameter for every attribute because, unlike constants and variables, attributes cannot have default values. As the following example shows, the \( n \)th parameter assigns a value to the \( n \)th attribute:

```
DECLARE
  r Rational;
BEGIN
  r := Rational(5, 6);  -- assign 5 to num, 6 to den
  -- now r is 5/6

  The next example shows that you can call a constructor using named notation instead of positional notation:

BEGIN
  r := Rational(den => 6, num => 5);  -- assign 5 to num, 6 to den
```

Calling Methods

Like packaged subprograms, methods are called using dot notation. In the following example, you call method normalize(), which divides attributes num and den (for "numerator" and "denominator") by their greatest common divisor:

```
DECLARE
  r Rational;
```
As the example below shows, you can chain method calls. Execution proceeds from left to right. First, member function \texttt{reciprocal()} is called, then member procedure \texttt{normalize()} is called.

```plsql
DECLARE
    r Rational := Rational(6, 8);
BEGIN
    r.reciprocal().normalize;
    dbms_output.put_line(r.num);  -- prints 4
END;
```

In SQL statements, calls to a parameterless method require an empty parameter list. In procedural statements, an empty parameter list is optional unless you chain calls, in which case it is required for all but the last call.

You cannot chain additional method calls to the right of a procedure call because a procedure is called as a statement, not as part of an expression. For example, the following statement is not allowed:

```plsql
r.normalize().reciprocal;  -- not allowed
```

Also, if you chain two function calls, the first function must return an object that can be passed to the second function.

For static methods, calls use the notation \texttt{type\_name.method\_name} rather than specifying an instance of the type.

When you call a method using an instance of a subtype, the actual method that is executed depends on the exact declarations in the type hierarchy. If the subtype overrides the method that it inherits from its supertype, the call uses the subtype’s implementation. Or, if the subtype does not override the method, the call uses the supertype’s implementation. This capability is known as \textit{dynamic method dispatch}.

### Sharing Objects

Most real-world objects are considerably larger and more complex than objects of type \texttt{Rational}. Consider the following object types:

```sql
CREATE TYPE Address AS OBJECT (
Sharing objects have twice as many attributes as Rational objects, and Person objects have still more attributes including one of type Address. When an object is large, it is inefficient to pass copies of it from subprogram to subprogram. It makes more sense to share the object. You can do that if the object has an object identifier. To share the object, you use references (refs for short). A ref is a pointer to an object.

Sharing has two important advantages. First, data is not replicated unnecessarily. Second, when a shared object is updated, the change occurs in only one place, and any ref can retrieve the updated values instantly.

In the following example, you gain the advantages of sharing by defining object type Home and then creating a table that stores instances of that object type:

```sql
CREATE TYPE Home AS OBJECT (
    address   VARCHAR2(35),
    owner     VARCHAR2(25),
    age       INTEGER,
    style     VARCHAR(15),
    floor plan BLOB,
    price     REAL(9,2),
    ...
);

CREATE TABLE homes OF Home;
```
Using Refs

By revising object type Person, you can model a community in which several people might share the same home. You use the type modifier REF to declare refs, which hold pointers to objects.

```
CREATE TYPE Person AS OBJECT (  
  first_name   VARCHAR2(10),
  last_name    VARCHAR2(15),
  birthday     DATE,
  home_address REF Home,  -- can be shared by family
  phone_number VARCHAR2(15),
  ss_number    INTEGER,
  mother       REF Person, -- family members refer to each other
  father       REF Person,
  ...
);
```

Notice how references from persons to homes and between persons model real-world relationships.

You can declare refs as variables, parameters, fields, or attributes. And, you can use refs as input or output variables in SQL data manipulation statements. However, you cannot navigate through refs. Given an expression such as `x.attribute`, where `x` is a ref, PL/SQL cannot navigate to the table in which the referenced object is stored. For example, the following assignment is illegal:

```
DECLARE  
  p_ref    REF Person;
  phone_no VARCHAR2(15);
BEGIN  
  phone_no := p_ref.phone_number;  -- illegal
END;
```

Instead, you must use the function DEREF or make calls to the package UTL_REF to access the object. For some examples, see "Using Function DEREF" on page 10-36.

Forward Type Definitions

You can refer only to schema objects that already exist. In the following example, the first CREATE TYPE statement is illegal because it refers to object type Department, which does not yet exist:

```
CREATE TYPE Employee AS OBJECT (  
  name VARCHAR2(20),
  dept REF Department,  -- illegal
  ...
);
```
Switching the CREATE TYPE statements does not help because the object types are mutually dependent. Object type Employee has an attribute that refers to object type Department, and object type Department has an attribute of type Employee. To solve this problem, you use a special CREATE TYPE statement called a forward type definition, which lets you define mutually dependent object types.

To debug the last example, simply precede it with the following statement:

```
CREATE TYPE Department;  -- forward type definition
-- at this point, Department is an incomplete object type
```

The object type created by a forward type definition is called an incomplete object type because (until it is defined fully) it has no attributes or methods.

An impure incomplete object type has attributes but causes compilation errors because it refers to an undefined type. For example, the following CREATE TYPE statement causes an error because object type Address is undefined:

```
CREATE TYPE Customer AS OBJECT (    
id NUMBER,                      
name VARCHAR2(20),             
addr Address,  -- not yet defined
phone VARCHAR2(15)            
);                            
```

This lets you defer the definition of object type Address. Moreover, the incomplete type Customer can be made available to other application developers for use in refs.

**Manipulating Objects**

You can use an object type in the CREATE TABLE statement to specify the datatype of a column. Once the table is created, you can use SQL statements to insert an object, select its attributes, call its methods, and update its state.

**Note:** Access to remote or distributed objects is not allowed.
In the SQL*Plus script below, the INSERT statement calls the constructor for object type Rational, then inserts the resulting object. The SELECT statement retrieves the value of attribute num. The UPDATE statement calls member method reciprocal(), which returns a Rational value after swapping attributes num and den. Notice that a table alias is required when you reference an attribute or method. (For an explanation, see Appendix D.)

```
CREATE TABLE numbers (rn Rational, ...)
/
INSERT INTO numbers (rn) VALUES (Rational(3, 62))  -- inserts 3/62
/
SELECT n.rn.num INTO my_num FROM numbers n ...  -- returns 3
/
UPDATE numbers n SET n.rn = n.rn.reciprocal() ...  -- yields 62/3
```

When you instantiate an object this way, it has no identity outside the database table. However, the object type exists independently of any table, and can be used to create objects in other ways.

In the next example, you create a table that stores objects of type Rational in its rows. Such tables, having rows of objects, are called object tables. Each column in a row corresponds to an attribute of the object type. Rows can have different column values.

```
CREATE TABLE rational_nums OF Rational;
```

Each row in an object table has an object identifier, which uniquely identifies the object stored in that row and serves as a reference to the object.

### Selecting Objects

Assume that you have run the following SQL*Plus script, which creates object type Person and object table persons, and that you have populated the table:

```
CREATE TYPE Person AS OBJECT (  
   first_name VARCHAR2(15),  
   last_name  VARCHAR2(15),  
   birthday   DATE,  
   home_address Address,  
   phone_number VARCHAR2(15))  
/
CREATE TABLE persons OF Person  
/
```
The following subquery produces a result set of rows containing only the attributes of Person objects:

BEGIN
    INSERT INTO employees -- another object table of type Person
        SELECT * FROM persons p WHERE p.last_name LIKE '%Smith';
END;

To return a result set of objects, you must use the function VALUE, which is discussed in the next section.

Using Function VALUE
As you might expect, the function VALUE returns the value of an object. VALUE takes as its argument a correlation variable. (In this context, a correlation variable is a row variable or table alias associated with a row in an object table.) For example, to return a result set of Person objects, use VALUE as follows:

BEGIN
    INSERT INTO employees
        SELECT VALUE(p) FROM persons p
            WHERE p.last_name LIKE '%Smith';
END;

In the next example, you use VALUE to return a specific Person object:

DECLARE
    p1 Person;
    p2 Person;
    ...
BEGIN
    SELECT VALUE(p) INTO p1 FROM persons p
        WHERE p.last_name = 'Kroll';
    p2 := p1;
    ...
END;

At this point, p1 holds a local Person object, which is a copy of the stored object whose last name is ‘Kroll’, and p2 holds another local Person object, which is a copy of p1. As the following example shows, you can use these variables to access and update the objects they hold:

BEGIN
    p1.last_name := p1.last_name || ' Jr';
END;

Now, the local Person object held by p1 has the last name ‘Kroll Jr’.
Using Function REF

You can retrieve refs using the function REF, which, like VALUE, takes as its argument a correlation variable. In the following example, you retrieve one or more refs to Person objects, then insert the refs into table person_refs:

```
BEGIN
  INSERT INTO person_refs
  SELECT REF(p) FROM persons p
  WHERE p.last_name LIKE '%Smith';
END;
```

In the next example, you retrieve a ref and attribute at the same time:

```
DECLARE
  p_ref       REF Person;
  taxpayer_id VARCHAR2(9);
BEGIN
  SELECT REF(p), p.ss_number INTO p_ref, taxpayer_id
  FROM persons p
  WHERE p.last_name = 'Parker'; -- must return one row
...
END;
```

In the final example, you update the attributes of a Person object:

```
DECLARE
  p_ref        REF Person;
  my_last_name VARCHAR2(15);
BEGIN
  SELECT REF(p) INTO p_ref FROM persons p
  WHERE p.last_name = my_last_name;
  UPDATE persons p
  SET p = Person('Jill', 'Anders', '11-NOV-67', ...)
  WHERE REF(p) = p_ref;
END;
```

Testing for Dangling Refs

If the object to which a ref points is deleted, the ref is left dangling (pointing to a nonexistent object). To test for this condition, you can use the SQL predicate IS DANGLING. For example, suppose column manager in relational table department holds refs to Employee objects stored in an object table. You can use the following UPDATE statement to convert any dangling refs into nulls:

```
UPDATE department SET manager = NULL WHERE manager IS DANGLING;
```
Using Function DEREF

You cannot navigate through refs within PL/SQL procedural statements. Instead, you must use the function `DEREF` in a SQL statement. (`DEREF` is short for dereference. When you dereference a pointer, you get the value to which it points.) `DEREF` takes as its argument a reference to an object, then returns the value of that object. If the ref is dangling, `DEREF` returns a null object.

In the example below, you dereference a ref to a `Person` object. Notice that you select the ref from dummy table `dual`. You need not specify an object table and search criteria because each object stored in an object table has a unique, immutable object identifier, which is part of every ref to that object.

```sql
DECLARE
    p1    Person;
    p_ref REF Person;
    name  VARCHAR2(15);
BEGIN
    ...  
    /* Assume that p_ref holds a valid reference to an object stored in an object table. */
    SELECT DEREF(p_ref) INTO p1 FROM dual;
    name := p1.last_name;
END;
```

You can use `DEREF` in successive SQL statements to dereference refs, as the following example shows:

```sql
CREATE TYPE PersonRef AS OBJECT (p_ref REF Person);
/
DECLARE
    name   VARCHAR2(15);
    pr_ref REF PersonRef;
    pr     PersonRef;
    p      Person;
BEGIN
    ...  
    /* Assume pr_ref holds a valid reference. */
    SELECT DEREF(pr_ref) INTO pr FROM dual;
    SELECT DEREF(pr.p_ref) INTO p FROM dual;
    name := p.last_name;
    ...  
END;
/
```

The next example shows that you cannot use function `DEREF` within procedural statements:
BEGIN

pl := DEREF(p_ref); -- not allowed

Within SQL statements, you can use dot notation to navigate through object columns to ref attributes and through one ref attribute to another. You can also navigate through ref columns to attributes if you use a table alias. For example, the following syntax is valid:

table_alias.object_column.ref_attribute

Assume that you have run the following SQL*Plus script, which creates object types Address and Person and object table persons:

CREATE TYPE Address AS OBJECT (
    street VARCHAR2(35),
    city VARCHAR2(15),
    state CHAR(2),
    zip_code INTEGER)
/

CREATE TYPE Person AS OBJECT (
    first_name VARCHAR2(15),
    last_name VARCHAR2(15),
    birthday DATE,
    home_address REF Address, -- shared with other Person objects
    phone_number VARCHAR2(15))
/

CREATE TABLE persons OF Person /

Ref attribute home_address corresponds to a column in object table persons that holds refs to Address objects stored in some other table. After populating the tables, you can select a particular address by dereferencing its ref, as follows:

DECLARE
    addr1 Address,
    addr2 Address,
...

BEGIN
    SELECT DEREF(home_address) INTO addr1 FROM persons p
    WHERE p.last_name = 'Derringer';
In the example below, you navigate through ref column `home_address` to attribute `street`. In this case, a table alias is required.

```sql
DECLARE
  my_street VARCHAR2(25),
  ...
BEGIN
  SELECT p.home_address.street INTO my_street FROM persons p
  WHERE p.last_name = 'Lucas';
END;
```

**Inserting Objects**

You use the `INSERT` statement to add objects to an object table. In the following example, you insert a `Person` object into object table `persons`:

```sql
BEGIN
  INSERT INTO persons
  VALUES ('Jenifer', 'Lapidus', ...);
END;
```

Alternatively, you can use the constructor for object type `Person` to insert an object into object table `persons`:

```sql
BEGIN
  INSERT INTO persons
  VALUES (Person('Albert', 'Brooker', ...));
END;
```

In the next example, you use the `RETURNING` clause to store `Person` refs in local variables. Notice how the clause mimics a `SELECT` statement. You can also use the `RETURNING` clause in `UPDATE` and `DELETE` statements.

```sql
DECLARE
  p1_ref REF Person;
  p2_ref REF Person;
BEGIN
  INSERT INTO persons p
  VALUES (Person('Paul', 'Chang', ...))
  RETURNING REF(p) INTO p1_ref;
  INSERT INTO persons p
  VALUES (Person('Ana', 'Thorne', ...))
  RETURNING REF(p) INTO p2_ref;
END;
```

To insert objects into an object table, you can use a subquery that returns objects of the same type. An example follows:

```sql
BEGIN
  INSERT INTO persons2
The rows copied to object table persons2 are given new object identifiers. No object identifiers are copied from object table persons.

The script below creates a relational table named department, which has a column of type Person, then inserts a row into the table. Notice how constructor Person() provides a value for column manager.

```
CREATE TABLE department (  
department VARCHAR2(20),  
manager Person,  
location VARCHAR2(20)) /  
INSERT INTO department  
VALUES ('Payroll', Person('Alan', 'Tsai', ...), 'Los Angeles') /  
```

The new Person object stored in column manager cannot be referenced because it is stored in a column (not a row) and therefore has no object identifier.

**Updating Objects**

To modify the attributes of objects in an object table, you use the UPDATE statement, as the following example shows:

```
BEGIN  
UPDATE persons p SET p.home_address = '341 Oakdene Ave'  
WHERE p.last_name = 'Brody';  
UPDATE persons p SET p = Person('Beth', 'Steinberg', ...)  
WHERE p.last_name = 'Steinway';  
END;
```

**Deleting Objects**

You use the DELETE statement to remove objects (rows) from an object table. To remove objects selectively, you use the WHERE clause, as follows:

```
BEGIN  
DELETE FROM persons p  
WHERE p.home_address = '108 Palm Dr';  
END;
```
This chapter shows you how to use native dynamic SQL (dynamic SQL for short), a PL/SQL interface that makes your applications more flexible and versatile. You learn simple ways to write programs that can build and process SQL statements “on the fly” at run time.

Within PL/SQL, you can execute any kind of SQL statement (even data definition and data control statements) without resorting to cumbersome programmatic approaches. Dynamic SQL blends seamlessly into your programs, making them more efficient, readable, and concise.

This chapter discusses the following topics:

- What Is Dynamic SQL?
- The Need for Dynamic SQL
- Using the EXECUTE IMMEDIATE Statement
- Using the OPEN-FOR, FETCH, and CLOSE Statements
- Tips and Traps for Dynamic SQL
What Is Dynamic SQL?

Most PL/SQL programs do a specific, predictable job. For example, a stored procedure might accept an employee number and salary increase, then update the `sal` column in the `emp` table. In this case, the full text of the `UPDATE` statement is known at compile time. Such statements do not change from execution to execution. So, they are called **static** SQL statements.

However, some programs must build and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different `SELECT` statements for the various reports it generates. In this case, the full text of the statement is unknown until run time. Such statements can, and probably will, change from execution to execution. So, they are called **dynamic** SQL statements.

Dynamic SQL statements are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments. A **placeholder** is an undeclared identifier, so its name, to which you must prefix a colon, does not matter. For example, PL/SQL makes no distinction between the following strings:

- `DELETE FROM emp WHERE sal > :my_sal AND comm < :my_comm`  
- `DELETE FROM emp WHERE sal > :s AND comm < :c`

To process most dynamic SQL statements, you use the `EXECUTE IMMEDIATE` statement. However, to process a multi-row query (`SELECT` statement), you must use the `OPEN-FOR`, `FETCH`, and `CLOSE` statements.

The Need for Dynamic SQL

You need dynamic SQL in the following situations:

- You want to execute a SQL data definition statement (such as `CREATE`), a data control statement (such as `GRANT`), or a session control statement (such as `ALTER SESSION`). In PL/SQL, such statements cannot be executed statically.

- You want more flexibility. For example, you might want to defer your choice of schema objects until run time. Or, you might want your program to build different search conditions for the `WHERE` clause of a `SELECT` statement. A more complex program might choose from various SQL operations, clauses, etc.

- You use package `DBMS_SQL` to execute SQL statements dynamically, but you want better performance, something easier to use, or functionality that `DBMS_SQL` lacks such as support for objects and collections. (For a comparison with `DBMS_SQL`, see Oracle9i Application Developer’s Guide - Fundamentals.)
Using the EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes a dynamic SQL statement or an anonymous PL/SQL block. The syntax is

```sql
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
  [, [IN | OUT | IN OUT] bind_argument]...]
[(RETURNING | RETURN) INTO bind_argument[, bind_argument]...];
```

where `dynamic_string` is a string expression that represents a SQL statement or PL/SQL block, `define_variable` is a variable that stores a selected column value, and `record` is a user-defined or `%ROWTYPE` record that stores a selected row. An input `bind_argument` is an expression whose value is passed to the dynamic SQL statement or PL/SQL block. An output `bind_argument` is a variable that stores a value returned by the dynamic SQL statement or PL/SQL block.

Except for multi-row queries, the dynamic string can contain any SQL statement (without the terminator) or any PL/SQL block (with the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement. For the right way, see "Making Procedures Work on Arbitrarily Named Schema Objects" on page 11-15.

Used only for single-row queries, the INTO clause specifies the variables or record into which column values are retrieved. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.

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

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

At run time, bind arguments replace corresponding placeholders in the dynamic string. So, every placeholder must be associated with a bind argument in the USING clause and/or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and...
NULL). To pass nulls to the dynamic string, you must use a workaround. See "Passing Nulls" on page 11-17.
Dynamic SQL supports all the SQL datatypes. So, for example, define variables and bind arguments can be collections, LOBs, instances of an object type, and refs. As a rule, dynamic SQL does not support PL/SQL-specific types. So, for example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the INTO clause.

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

**Some Examples of Dynamic SQL**

The following PL/SQL block contains several examples of dynamic SQL:

```plsql
DECLARE
  sql_stmt VARCHAR2(200);
  plsql_block VARCHAR2(500);
  emp_id NUMBER(4) := 7566;
  salary NUMBER(7,2);
  dept_id NUMBER(2) := 50;
  dept_name VARCHAR2(14) := 'PERSONNEL';
  location VARCHAR2(13) := 'DALLAS';
  emp_rec emp%ROWTYPE;
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';
  sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
  EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;
  sql_stmt := 'SELECT * FROM emp WHERE empno = :1';
  EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;
  plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
  EXECUTE IMMEDIATE plsql_block USING 7788, 500;
  sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
                RETURNING sal INTO :2';
  EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;
  EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
                    USING dept_id;
  EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
END;
```
In the example below, a standalone procedure accepts the name of a database table (such as ‘emp’) and an optional WHERE-clause condition (such as ‘sal > 2000’). If you omit the condition, the procedure deletes all rows from the table. Otherwise, the procedure deletes only those rows that meet the condition.

```sql
CREATE PROCEDURE delete_rows (  
    table_name IN VARCHAR2,  
    condition IN VARCHAR2 DEFAULT NULL) AS
  where_clause VARCHAR2(100) := ' WHERE ' || condition;
BEGIN  
  IF condition IS NULL THEN where_clause := NULL; END IF;
  EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || where_clause;
EXCEPTION
  ...  
END;
```

**Backward Compatibility of the USING Clause**

When a dynamic INSERT, UPDATE, or DELETE statement has a RETURNING clause, output bind arguments can go in the RETURNING INTO clause or the USING clause. In new applications, use the RETURNING INTO clause. In old applications, you can continue to use the USING clause. For example, both of the following EXECUTE IMMEDIATE statements are legal:

```sql
DECLARE
  sql_stmt VARCHAR2(200);
  my_empno NUMBER(4) := 7902;
  my_ename VARCHAR2(10);
  my_job   VARCHAR2(9);
  my_sal   NUMBER(7,2) := 3250.00;
BEGIN
  sql_stmt := 'UPDATE emp SET sal = :1 WHERE empno = :2  
               RETURNING ename, job INTO :3, :4';

  /* Bind returned values through USING clause. */
  EXECUTE IMMEDIATE sqlStmt
    USING my_sal, my_empno, OUT my_ename, OUT my_job;

  /* Bind returned values through RETURNING INTO clause. */
  EXECUTE IMMEDIATE sqlStmt
    USING my_sal, my_empno RETURNING INTO my_ename, my_job;
...
END;
```
Specifying Parameter Modes

With the USING clause, you need not specify a parameter mode for input bind arguments because the mode defaults to IN. With the RETURNING INTO clause, you cannot specify a parameter mode for output bind arguments because, by definition, the mode is OUT. An example follows:

DECLARE
  sql_stmt VARCHAR2(200);
  dept_id NUMBER(2) := 30;
  old_loc VARCHAR2(13);
BEGIN
  sql_stmt :=
    'DELETE FROM dept WHERE deptno = :1 RETURNING loc INTO :2';
  EXECUTE IMMEDIATE sql_stmt USING dept_id RETURNING INTO old_loc;
  ...
END;

When appropriate, you must specify the OUT or IN OUT mode for bind arguments passed as parameters. For example, suppose you want to call the following standalone procedure:

CREATE PROCEDURE create_dept ( 
  deptno IN OUT NUMBER, 
  dname IN VARCHAR2, 
  loc IN VARCHAR2) AS 
BEGIN 
  SELECT deptno_seq.NEXTVAL INTO deptno FROM dual;
  INSERT INTO dept VALUES (deptno, dname, loc);
END;

To call the procedure from a dynamic PL/SQL block, you must specify the IN OUT mode for the bind argument associated with formal parameter deptno, as follows:

DECLARE
  plsql_block VARCHAR2(500);
  new_deptno NUMBER(2);
  new_dname VARCHAR2(14) := 'ADVERTISING';
  new_loc VARCHAR2(13) := 'NEW YORK';
BEGIN
  plsql_block := 'BEGIN create_dept(:a, :b, :c); END;';
  EXECUTE IMMEDIATE plsql_block 
    USING IN OUT new_deptno, new_dname, new_loc;
  IF new_deptno > 90 THEN ...
END;
Using the OPEN-FOR, FETCH, and CLOSE Statements

You use three statements to process a dynamic multi-row query: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set one at a time. When all the rows are processed, you CLOSE the cursor variable. (For more information about cursor variables, see "Using Cursor Variables" on page 6-16.)

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi-row query, executes the query, identifies the result set, positions the cursor on the first row in the result set, then zeroes the rows-processed count kept by %ROWCOUNT.

Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is

```
OPEN {cursor_variable | :host_cursor_variable} FOR dynamic_string
[USING bind_argument[, bind_argument]...];
```

where cursor_variable is a weakly typed cursor variable (one without a return type), host_cursor_variable is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multi-row query.

In the following example, you declare a cursor variable, then associate it with a dynamic SELECT statement that returns rows from the emp table:

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR; -- define weak REF CURSOR type
    emp_cv   EMPCurTyp; -- declare cursor variable
    my_ename VARCHAR2(15);
    my_sal   NUMBER := 1000;
BEGIN
    OPEN emp_cv FOR -- open cursor variable
        'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
    ...
END;
```

Any bind arguments in the query are evaluated only when the cursor variable is opened. So, to fetch from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.
Fetching from the Cursor Variable

The `FETCH` statement returns a row from the result set of a multi-row query, assigns the values of select-list items to corresponding variables or fields in the `INTO` clause, increments the count kept by `%ROWCOUNT`, and advances the cursor to the next row.

The syntax follows:

```
FETCH {cursor_variable | :host_cursor_variable}
    INTO {define_variable[, define_variable]... | record};
```

Continuing the example, you fetch rows from cursor variable `emp_cv` into define variables `my_ename` and `my_sal`:

```
LOOP
    FETCH emp_cv INTO my_ename, my_sal; -- fetch next row
    EXIT WHEN emp_cv%NOTFOUND; -- exit loop when last row is fetched
    -- process row
END LOOP;
```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible variable or field in the `INTO` clause. 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.

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Closing the Cursor Variable

The `CLOSE` statement disables a cursor variable. After that, the associated result set is undefined. The syntax follows:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this example, when the last row is processed, you close cursor variable `emp_cv`:

```
LOOP
    FETCH emp_cv INTO my_ename, my_sal;
    EXIT WHEN emp_cv%NOTFOUND;
    -- process row
END LOOP;
CLOSE emp_cv; -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises `INVALID_CURSOR`. 

Examples of Dynamic SQL for Records, Objects, and Collections

As the following example shows, you can fetch rows from the result set of a dynamic multi-row query into a record:

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv   EmpCurTyp;
    emp_rec  emp%ROWTYPE;
    sql_stmt VARCHAR2(200);
    my_job   VARCHAR2(15) := 'CLERK';
BEGIN
    sql_stmt := 'SELECT * FROM emp WHERE job = :j';
    OPEN emp_cv FOR sql_stmt USING my_job;
    LOOP
        FETCH emp_cv INTO emp_rec;
        EXIT WHEN emp_cv%NOTFOUND;
    /* process record */
    END LOOP;
    CLOSE emp_cv;
END;
```

The next example illustrates the use of objects and collections. Suppose you define object type `Person` and `VARRAY` type `Hobbies`, as follows:

```
CREATE TYPE Person AS OBJECT (name VARCHAR2(25), age NUMBER);
CREATE TYPE Hobbies IS VARRAY(10) OF VARCHAR2(25);
```

Now, using dynamic SQL, you can write a package of procedures that uses these types, as follows:

```
CREATE PACKAGE teams AS
    PROCEDURE create_table (tab_name VARCHAR2);
    PROCEDURE insert_row (tab_name VARCHAR2, p Person, h Hobbies);
    PROCEDURE print_table (tab_name VARCHAR2);
END;
```

```
CREATE PACKAGE BODY teams AS
    PROCEDURE create_table (tab_name VARCHAR2) IS
        BEGIN
            EXECUTE IMMEDIATE 'CREATE TABLE ' || tab_name || ' (pers Person, hobbs Hobbies)';
        END;
```
PROCEDURE insert_row (tab_name VARCHAR2, p Person, h Hobbies) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO ' || tab_name || ' VALUES (:1, :2)' USING p, h;
END;

PROCEDURE print_table (tab_name VARCHAR2) IS
  TYPE RefCurTyp IS REF CURSOR;
  cv RefCurTyp;
  p Person;
  h Hobbies;
BEGIN
  OPEN cv FOR 'SELECT pers, hobbs FROM ' || tab_name;
  LOOP
    FETCH cv INTO p, h;
    EXIT WHEN cv%NOTFOUND;
    -- print attributes of 'p' and elements of 'h'
  END LOOP;
  CLOSE cv;
END;
END;

From an anonymous PL/SQL block, you might call the procedures in package teams, as follows:

DECLARE
  team_name VARCHAR2(15);
  ...
BEGIN
  ...
  team_name := 'Notables';
  teams.create_table(team_name);
  teams.insert_row(team_name, Person('John', 31),
       Hobbies('skiing', 'coin collecting', 'tennis'));
  teams.insert_row(team_name, Person('Mary', 28),
       Hobbies('golf', 'quilting', 'rock climbing'));
  teams.print_table(team_name);
END;
Using Bulk Dynamic SQL

In this section, you learn how to add the power of bulk binding to dynamic SQL. Bulk binding improves performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binding, entire collections, not just individual elements, are passed back and forth.

Using the following commands, clauses, and cursor attribute, your applications can construct bulk SQL statements, then execute them dynamically at run time:

- BULK FETCH statement
- BULK EXECUTE IMMEDIATE statement
- FORALL statement
- COLLECT INTO clause
- RETURNING INTO clause
- %BULK_ROWCOUNT cursor attribute

The static versions of these statements, clauses, and cursor attribute are discussed in "Reducing Loop Overhead for Collections with Bulk Binds" on page 5-32. Refer to that section for background information.

Syntax for Dynamic Bulk Binds

Bulk binding lets Oracle bind a variable in a SQL statement to a collection of values. The collection type can be any PL/SQL collection type (index-by table, nested table, or varray). However, the collection elements must have a SQL datatype such as CHAR, DATE, or NUMBER. Three statements support dynamic bulk binds: EXECUTE IMMEDIATE, FETCH, and FORALL.

**Bulk EXECUTE IMMEDIATE**

This statement lets you bulk-bind define variables or OUT bind arguments passed as parameters to a dynamic SQL statement. The syntax follows:

```
EXECUTE IMMEDIATE dynamic_string
    [[BULK COLLECT] INTO define_variable[, define_variable ...]]
    [USING bind_argument[, bind_argument ...]]
    [{RETURNING | RETURN}
    BULK COLLECT INTO bind_argument[, bind_argument ...]]; 
```

With a dynamic multi-row query, you can use the BULK COLLECT INTO clause to bind define variables. The values in each column are stored in a collection.
With a dynamic INSERT, UPDATE, or DELETE statement that returns multiple rows, you can use the RETURNING BULK COLLECT INTO clause to bulk-bind output variables. The returned rows of values are stored in a set of collections.

**Bulk FETCH**

This statement lets you fetch from a dynamic cursor the same way you fetch from a static cursor. The syntax follows:

```sql
FETCH dynamic_cursor
    BULK COLLECT INTO define_variable[, define_variable ...];
```

If the number of define variables in the BULK COLLECT INTO list exceeds the number of columns in the query select-list, Oracle generates an error.

**Bulk FORALL**

This statement lets you bulk-bind input variables in a dynamic SQL statement. In addition, you can use the EXECUTE IMMEDIATE statement inside a FORALL loop. The syntax follows:

```sql
FORALL index IN lower bound..upper bound
    EXECUTE IMMEDIATE dynamic_string
        USING bind_argument | bind_argument(index)
        [, bind_argument | bind_argument(index)] ...
    [(RETURNING | RETURN) BULK COLLECT
        INTO bind_argument[, bind_argument ... ]];
```

The dynamic string must represent an INSERT, UPDATE, or DELETE statement (not a SELECT statement).

**Examples of Dynamic Bulk Binds**

You can bind define variables in a dynamic query using the BULK COLLECT INTO clause. As the following example shows, you can use that clause in a bulk FETCH or bulk EXECUTE IMMEDIATE statement:

```sql
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    TYPE NumList IS TABLE OF NUMBER;
    TYPE NameList IS TABLE OF VARCHAR2(15);
    emp_cv EmpCurTyp;
    empnos NumList;
    enames NameList;
    sals   NumList;
```
Using Bulk Dynamic SQL

BEGIN
    OPEN emp_cv FOR 'SELECT empno, ename FROM emp';
    FETCH emp_cv BULK COLLECT INTO empnos, enames;
    CLOSE emp_cv;

    EXECUTE IMMEDIATE 'SELECT sal FROM emp'
        BULK COLLECT INTO sals;
END;

Only the INSERT, UPDATE, and DELETE statements can have output bind variables. To bulk-bind them, you use the BULK RETURNING INTO clause, which can appear only in an EXECUTE IMMEDIATE. An example follows:

DECLARE
    TYPE NameList IS TABLE OF VARCHAR2(15);
    enames    NameList;
    bonus_amt NUMBER := 500;
    sql_stmt  VARCHAR(200);
BEGIN
    sql_stmt := 'UPDATE emp SET bonus = :1 RETURNING ename INTO :2';
    EXECUTE IMMEDIATE sql_stmt
        USING bonus_amt RETURNING BULK COLLECT INTO enames;
END;

To bind the input variables in a SQL statement, you can use the FORALL statement and USING clause, as shown below. However, the SQL statement cannot be a query.

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    TYPE NameList IS TABLE OF VARCHAR2(15);
    empnos NumList;
    enames NameList;
BEGIN
    empnos := NumList(1,2,3,4,5);
    FORALL i IN 1..5
      EXECUTE IMMEDIATE
        'UPDATE emp SET sal = sal * 1.1 WHERE empno = :1
         RETURNING ename INTO :2'
        USING empnos(i) RETURNING BULK COLLECT INTO enames;
    ...
END;
Tips and Traps for Dynamic SQL

This section shows you how to take full advantage of dynamic SQL and how to avoid some common pitfalls.

Improving Performance

In the example below, Oracle opens a different cursor for each distinct value of emp_id. This can lead to resource contention and poor performance.

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
   EXECUTE IMMEDIATE
      'DELETE FROM emp WHERE empno = ' || TO_CHAR(emp_id);
END;
```

You can improve performance by using a bind variable, as shown below. This allows Oracle to reuse the same cursor for different values of emp_id.

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
   EXECUTE IMMEDIATE
      'DELETE FROM emp WHERE empno = :num' USING emp_id;
END;
```

Making Procedures Work on Arbitrarily Named Schema Objects

Suppose you need a procedure that accepts the name of any database table, then drops that table from your schema. Using dynamic SQL, you might write the following standalone procedure:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE :tab' USING table_name;
END;
```

However, at run time, this procedure fails with an invalid table name error. That is because you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement. Instead, you must embed parameters in the dynamic string, then pass the names of schema objects to those parameters.
To debug the last example, you must revise the EXECUTE IMMEDIATE statement. Instead of using a placeholder and bind argument, you embed parameter table_name in the dynamic string, as follows:

CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
    BEGIN
        EXECUTE IMMEDIATE 'DROP TABLE ' || table_name;
    END;

Now, you can pass the name of any database table to the dynamic SQL statement.

**Using Duplicate Placeholders**

Placeholders in a dynamic SQL statement are associated with bind arguments in the USING clause by position, not by name. So, if the same placeholder appears two or more times in the SQL statement, each appearance must correspond to a bind argument in the USING clause. For example, given the dynamic string

sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';

you might code the corresponding USING clause as follows:

EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;

However, only the unique placeholders in a dynamic PL/SQL block are associated with bind arguments in the USING clause by position. So, if the same placeholder appears two or more times in a PL/SQL block, all appearances correspond to one bind argument in the USING clause. In the example below, the first unique placeholder (x) is associated with the first bind argument (a). Likewise, the second unique placeholder (y) is associated with the second bind argument (b).

DECLARE
    a NUMBER := 4;
    b NUMBER := 7;
BEGIN
    plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;'
    EXECUTE IMMEDIATE plsql_block USING a, b;
...
END;
Using Cursor Attributes

Every explicit cursor has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor name, they return useful information about the execution of static and dynamic SQL statements.

To process SQL data manipulation statements, Oracle opens an implicit cursor named SQL. Its attributes return information about the most recently executed INSERT, UPDATE, DELETE, or single-row SELECT statement. For example, the following standalone function uses %ROWCOUNT to return the number of rows deleted from a database table:

```sql
CREATE FUNCTION rows_deleted (table_name IN VARCHAR2, condition IN VARCHAR2) RETURN INTEGER AS
BEGIN
    EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || ' WHERE ' || condition;
    RETURN SQL%ROWCOUNT;  -- return number of rows deleted
END;
```

Likewise, when appended to a cursor variable name, the cursor attributes return information about the execution of a multi-row query. For more information about cursor attributes, see "Using Cursor Attributes" on page 6-34.

Passing Nulls

Suppose you want to pass nulls to a dynamic SQL statement. For example, you might write the following EXECUTE IMMEDIATE statement:

```sql
EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING NULL;
```

However, this statement fails with a bad expression error because the literal NULL is not allowed in the USING clause. To work around this restriction, simply replace the keyword NULL with an uninitialized variable, as follows:

```sql
DECLARE
    a_null CHAR(1);  -- set to NULL automatically at run time
BEGIN
    EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING a_null;
END;
```
Doing Remote Operations

As the following example shows, PL/SQL subprograms can execute dynamic SQL statements that refer to objects on a remote database:

```sql
PROCEDURE delete_dept (db_link VARCHAR2, dept_id INTEGER) IS
    BEGIN
        EXECUTE IMMEDIATE 'DELETE FROM dept@' || db_link ||
        ' WHERE deptno = :num' USING dept_id;
    END;
```

Also, the targets of remote procedure calls (RPCs) can contain dynamic SQL statements. For example, suppose the following standalone function, which returns the number of rows in a table, resides on the Chicago database:

```sql
CREATE FUNCTION row_count (tab_name VARCHAR2) RETURN INTEGER AS
    rows INTEGER;
    BEGIN
        EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || tab_name INTO rows;
        RETURN rows;
    END;
```

From an anonymous block, you might call the function remotely, as follows:

```sql
DECLARE
    emp_count INTEGER;
BEGIN
    emp_count := row_count@chicago('emp');
```

Using Invoker Rights

By default, a stored procedure executes with the privileges of its definer, not its invoker. Such procedures are bound to the schema in which they reside. For example, assume that the following standalone procedure, which can drop any kind of database object, resides in schema `scott`:

```sql
CREATE PROCEDURE drop_it (kind IN VARCHAR2, name IN VARCHAR2) AS
    BEGIN
        EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
    END;
```

Also assume that user `jones` has been granted the EXECUTE privilege on this procedure. When user `jones` calls `drop_it`, as follows, the dynamic DROP statement executes with the privileges of user `scott`:

```sql
SQL> CALL drop_it('TABLE', 'dept');
```
Also, the unqualified reference to table dept is resolved in schema scott. So, the procedure drops the table from schema scott, not from schema jones.

However, the AUTHID clause enables a stored procedure to execute with the privileges of its invoker (current user). Such procedures are not bound to a particular schema. For example, the following version of drop_it executes with the privileges of its invoker:

```sql
CREATE PROCEDURE drop_it (kind IN VARCHAR2, name IN VARCHAR2)
    AUTHID CURRENT_USER AS
BEGIN
    EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
END;
```

Also, the unqualified reference to the database object is resolved in the schema of the invoker. For details, see “Invoker Rights Versus Definer Rights” on page 8-47.

**Using Pragma RESTRICT_REFERENCES**

A function called from SQL statements must obey certain rules meant to control side effects. (See "Controlling Side Effects of PL/SQL Subprograms" on page 8-9.) To check for violations of the rules, you can use the pragma RESTRICT_REFERENCES. The pragma asserts that a function does not read and/or write database tables and/or package variables. (For more information, See Oracle9i Application Developer’s Guide - Fundamentals.)

However, if the function body contains a dynamic INSERT, UPDATE, or DELETE statement, the function always violates the rules "write no database state" (WNDS) and "read no database state" (RNDS). That is because dynamic SQL statements are checked at run time, not at compile time. In an EXECUTE IMMEDIATE statement, only the INTO clause can be checked at compile time for violations of RNDS.

**Avoiding Deadlocks**

In a few situations, executing a SQL data definition statement results in a deadlock. For example, the procedure below causes a deadlock because it attempts to drop itself. To avoid deadlocks, never try to ALTER or DROP a subprogram or package while you are still using it.

```sql
CREATE PROCEDURE calc_bonus (emp_id NUMBER) AS
BEGIN
...
    EXECUTE IMMEDIATE 'DROP PROCEDURE calc_bonus';
END;
```
Tips and Traps for Dynamic SQL
Every day, in every way, I am getting better and better. —Émile Coué

Over time, even the performance of well-designed applications can degrade. So, periodic tuning is an important part of application maintenance. This chapter shows you how to make small adjustments that improve performance. By tuning your applications, you can make sure they continue to deliver the required response time and throughput.

This chapter discusses the following topics:

- Reasons for PL/SQL Performance Problems
- Identifying PL/SQL Performance Problems
- PL/SQL Features for Performance Tuning
Reasons for 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.

Badly Written SQL Statements in a PL/SQL Program

PL/SQL programs look relatively simple because their complexity is hidden in SQL statements, which do most of the work. That is why badly written SQL statements are the main reason for slow execution. If a program contains many badly written SQL statements, no number of well-written PL/SQL statements will help.

If badly written SQL statements are slowing down your program, analyze their execution plans and performance using the methods listed below. Then, rewrite the SQL statements. For example, hints to the query optimizer can eliminate problems such as unnecessary full-table scans.

- EXPLAIN PLAN statement
- SQL Trace facility with TKPROF utility
- Oracle Trace facility

For more information about these methods, see Oracle9i Database Performance Methods.

Poor Programming Practices

Often, poor programming practices are a side effect of schedule crunches. In such circumstances, even experienced programmers might write code that hampers performance.

No matter how suitable a programming language is for a given task, badly written subprograms (for example, a slow sort or search function) can ruin performance. Suppose the subprogram called most often by an application is a lookup function with hundreds of possible targets. If that function could be written as a hash or a binary search but, instead, is written as a linear search, overall performance suffers.

Other poor practices include declaring variables that are never used, passing unneeded parameters to functions and procedures, placing initializations or computations inside a loop needlessly, and so on.

Duplication of Built-in Functions

PL/SQL provides many highly optimized functions such as REPLACE, TRANSLATE, SUBSTR, INSTR, RPAD, and LTRIM. Do not hand code your own versions. Built-in
functions are more efficient. Even when a built-in function has more power than you need, use it rather than hand-coding a subset of its functionality.

**Inefficient Conditional Control Statements**

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. For example, in the following OR expression, when the value of `sal` is less than 1500, the left operand yields TRUE, so PL/SQL need not evaluate the right operand (because OR returns TRUE if either of its operands is true):

```
IF (sal < 1500) OR (comm IS NULL) THEN
    ...
END IF;
```

Now, consider the following AND expression:

```
IF credit_ok(cust_id) AND (loan < 5000) THEN
    ...
END IF;
```

The Boolean function credit_ok is always called. However, if you switch the operands of AND as follows

```
IF (loan < 5000) AND credit_ok(cust_id) THEN
    ...
END IF;
```

the function is called only when the expression `loan < 5000` is true (because AND returns TRUE only if both its operands are true).

The same idea applies to EXIT-WHEN statements.

**Implicit Datatype Conversions**

At run time, PL/SQL converts between structurally different datatypes implicitly. For instance, assigning a `PLS_INTEGER` variable to a `NUMBER` variable results in a conversion because their internal representations are different.

Avoiding implicit conversions can improve performance. Look at the example below. The integer literal `15` is represented internally as a signed 4-byte quantity, so PL/SQL must convert it to an Oracle number before the addition. However, the floating-point literal `15.0` is represented as a 22-byte Oracle number, so no conversion is necessary.

```
DECLARE
    ...
END;
```
Reasons for PL/SQL Performance Problems

n NUMBER;
c CHAR(5);
BEGIN
    n := n + 15;    -- converted
    n := n + 15.0;  -- not converted
...
END;

Here is another example:

DECLARE
c CHAR(5);
BEGIN
c := 25;    -- converted
c := '25';  -- not converted
...
END;

Inappropriate Declarations for Numeric Datatypes

The datatype NUMBER and its subtypes are 22-byte, database-format numbers, designed for portability and arbitrary scale/precision, not performance. When you need to declare an integer variable, use the datatype PLS_INTEGER, which is the most efficient numeric type. That is because PLS_INTEGER values require less storage than INTEGER or NUMBER values. Also, PLS_INTEGER operations use machine arithmetic, so they are faster than BINARY_INTEGER, INTEGER, or NUMBER operations, which use library arithmetic.

Furthermore, INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNTYPE are constrained subtypes. So, their variables require precision checking at run time, which can affect performance.

Unnecessary NOT NULL Constraints

In PL/SQL, using the NOT NULL constraint incurs a performance cost. Consider the following example:

PROCEDURE calc_m IS
    m NUMBER NOT NULL := 0;
a NUMBER;
b NUMBER;
BEGIN
...
    m := a + b;
...
END;
Because \( m \) is constrained by \texttt{NOT NULL}, the value of the expression \( a + b \) is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to \( m \). Otherwise, an exception is raised. However, if \( m \) were not constrained, the value would be assigned to \( m \) directly.

A more efficient way to write the last example follows:

```plsql
PROCEDURE calc_m IS
  m NUMBER;  -- no constraint
  a NUMBER;
  b NUMBER;
BEGIN
  ...
  m := a + b;
  IF m IS NULL THEN  -- enforce constraint programmatically
    ...
  END IF;
END;
```

Note that the subtypes \texttt{NATURALN} and \texttt{POSTIVEN} are defined as \texttt{NOT NULL}. So, using them incurs the same performance cost.

**Size Declarations for VARCHAR2 Variables**

The \texttt{VARCHAR2} datatype involves a trade-off between memory use and efficiency. For a \texttt{VARCHAR2(\geq 2000)} variable, PL/SQL dynamically allocates only enough memory to hold the actual value. However, for a \texttt{VARCHAR2(< 2000)} variable, PL/SQL preallocates enough memory to hold a maximum-size value. So, for example, if you assign the same 500-byte value to a \texttt{VARCHAR2(2000)} variable and to a \texttt{VARCHAR2(1999)} variable, the latter uses 1499 bytes more memory.

**Misuse of Shared Memory in a PL/SQL Program**

When you call a packaged subprogram for the first time, the whole package is loaded into the shared memory pool. So, subsequent calls to related subprograms in the package require no disk I/O, and your code executes faster. However, if the package is aged out of memory, it must be reloaded if you reference it again.

You can improve performance by sizing the shared memory pool correctly. Make sure it is large enough to hold all frequently used packages but not so large that memory is wasted.
Pinned Packages

Another way to improve performance is to pin frequently used packages in the shared memory pool. When a package is pinned, it is not aged out by the least recently used (LRU) algorithm that Oracle normally uses. The package remains in memory no matter how full the pool gets or how frequently you access the package.

You can pin packages with the help of the supplied package `DBMS_SHARED_POOL`. For more information, see *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Serially Reusable Packages

To help you manage the use of memory, PL/SQL provides the pragma `SERIALLY_REUSABLE`, which lets you mark some packages as *serially reusable*. You can so mark a package if its state is needed only for the duration of one call to the server (for example, an OCI call to the server or a server-to-server RPC).

The global memory for such 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`.

The maximum number of work areas needed for a package is the number of concurrent users of that package, which is usually much smaller than the number of logged-on users. The increased use of SGA memory is more than offset by the decreased use of UGA memory. Also, Oracle ages-out work areas not in use if it needs to reclaim SGA memory.
For bodiless packages, you code the pragma in the package spec using the following syntax:

```sql
PRAGMA SERIALLY_REUSABLE;
```

For packages with a body, you must code the pragma in the spec and body. You cannot code the pragma only in the body. The following example shows how a public variable in a serially reusable package behaves across call boundaries:

```sql
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;
/
BEGIN
    /* Initialize package state. */
    pkg1.init_pkg_state(4);
    /* On same server call, print package state. */
    pkg1.print_pkg_state;  -- prints 4
END;
/
-- subsequent server call
BEGIN
    -- the package’s public variable is initialized
    -- to the default value automatically
    pkg1.print_pkg_state;  -- prints 0
END;
```

For more information, see *Oracle9i Application Developer’s Guide - Fundamentals.*
Identifying PL/SQL Performance Problems

As you develop larger and larger PL/SQL applications, it becomes more difficult to isolate performance problems. So, PL/SQL provides a Profiler API to profile run-time behavior and to help you identify performance bottlenecks. PL/SQL also provides a Trace API for tracing the execution of programs on the server. You can use Trace to trace the execution by subprogram or exception.

The Profiler API: Package DBMS_PROFILER

The Profiler API is implemented as PL/SQL package DBMS_PROFILER, which provides services for gathering and saving run-time statistics. The information is stored in database tables, which you can query later. For example, you can learn how much time was spent executing each PL/SQL line and subprogram.

To use the Profiler, you start the profiling session, run your application long enough to get adequate code coverage, flush the collected data to the database, then stop the profiling session. In a typical session, you take the following steps:

4. Start by calling the procedure start_profiler in package DBMS_PROFILER and associating a comment with the Profiler session.
5. Run the application to be profiled.
6. Call the procedure flush_data repeatedly to save incremental data and free memory allocated for data structures.
7. Stop by calling the procedure stop_profiler.

The Profiler traces the execution of your program, computing the time spent at each line and in each subprogram. You can use the collected data to improve performance. For instance, you might focus on subprograms that run slowly.

For more information about package DBMS_PROFILER, see Oracle9i Supplied PL/SQL Packages and Types Reference.

Analyzing the Collected Performance Data

The next step is to determine why more time was spent executing certain code segments or accessing certain data structures. Find the problem areas by querying the performance data. Focus on the subprograms and packages that use up the most execution time, inspecting possible performance bottlenecks such as SQL statements, loops, and recursive functions.
Using Trace Data to Improve Performance

Use the results of your analysis to rework slow algorithms. For example, due to an exponential growth in data, you might need to replace a linear search with a binary search. Also, look for inefficiencies caused by inappropriate data structures, and, if necessary, replace those data structures.

The Trace API: Package DBMS_TRACE

With large, complex applications, it becomes difficult to keep track of calls between subprograms. By tracing your code with the Trace API, you can see the order in which subprograms execute. The Trace API is implemented as PL/SQL package DBMS_TRACE, which provides services for tracing execution by subprogram or exception.

To use Trace, you start the tracing session, run your application, then stop the tracing session. As the program executes, trace data is collected and stored in database tables. In a typical session, you take the following steps:

1. Optionally, select specific subprograms for trace data collection.
2. Start by calling the procedure set_plsql_trace in package DBMS_TRACE.
3. Run the application to be traced.
4. Stop by calling the procedure clear_plsql_trace.

For more information about package DBMS_TRACE, see Oracle9i Supplied PL/SQL Packages and Types Reference.

Controlling the Trace

Tracing large applications can produce huge amounts of data that are difficult to manage. Before starting Trace, you can optionally limit the volume of data collected by selecting specific subprograms for trace data collection.

In addition, you can choose a tracing level. For example, you can choose to trace all subprograms and exceptions, or you can choose to trace selected subprograms and exceptions.
PL/SQL Features for Performance Tuning

After correcting the flaws that slow down an application, you can use the following PL/SQL features and techniques:

- Improving PL/SQL Performance with Native Dynamic SQL
- Improving PL/SQL Performance with Bulk Binds
- Improving PL/SQL Performance with the NOCOPY Compiler Hint
- Improving PL/SQL Performance with the RETURNING Clause
- Improving PL/SQL Performance with External Routines
- Improving PL/SQL Performance with Object Types and Collections
- Compiling PL/SQL Code for Native Execution

These easy-to-use features can speed up an application considerably.

Improving PL/SQL Performance with Native Dynamic SQL

Some programs (a general-purpose report writer for example) must build and process a variety of SQL statements at run time. So, their full text is unknown until then. Such statements can, and probably will, change from execution to execution. So, 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.

Native dynamic SQL is easier to use and much faster than the DBMS_SQL package. In the following example, you declare a cursor variable, then associate it with a dynamic SELECT statement that returns rows from database table emp:

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv   EmpCurTyp;
    my_ename VARCHAR2(15);
    my_sal   NUMBER := 1000;
BEGIN
    OPEN emp_cv FOR
        'SELECT ename, sal FROM emp
         WHERE sal > :s' USING my_sal;
    ...
END;
```
For more information, see Chapter 11.

Improving PL/SQL Performance with Bulk Binds

When SQL statements execute inside a loop using collection elements as bind variables, context switching between the PL/SQL and SQL engines can slow down execution. For example, the following `UPDATE` statement is sent to the SQL engine with each iteration of the `FOR` loop:

```sql
DECLARE
    TYPE NumList IS VARRAY(20) OF NUMBER;
    depts NumList := NumList(10, 30, 70, ...);  -- department numbers
BEGIN
    ...
    FOR i IN depts.FIRST..depts.LAST LOOP
        ...
        UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
    END LOOP;
END;
```

In such cases, if the SQL statement affects four or more database rows, the use of bulk binds can improve performance considerably. For example, the following `UPDATE` statement is sent to the SQL engine just once, with the entire nested table:

```sql
FORALL i IN depts.FIRST..depts.LAST
    UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
```

To maximize performance, rewrite your programs as follows:

- If an `INSERT`, `UPDATE`, or `DELETE` statement executes inside a loop and references collection elements, move it into a `FORALL` statement.
- If a `SELECT INTO`, `FETCH INTO`, or `RETURNING INTO` clause references a collection, incorporate the `BULK COLLECT` clause.
- If possible, use host arrays to pass collections back and forth between your programs and the database server.
- If the failure of a DML operation on a particular row is not a serious problem, include the keywords `SAVE EXCEPTIONS` in the `FORALL` statement and report or clean up the errors in a subsequent loop using the `%BULK_EXCEPTIONS` attribute.

These are not a trivial tasks. They require careful analysis of program control-flows and dependencies.
For more information about bulk binding, see "Reducing Loop Overhead for Collections with Bulk Binds" on page 5-32.

Improving PL/SQL Performance with the NOCOPY Compiler Hint

By default, OUT and IN OUT parameters are passed by value. That is, the value of an IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory. To prevent that, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. In the following example, you ask the compiler to pass IN OUT parameter my_unit by reference instead of by value:

```plsql
DECLARE
    TYPE Platoon IS VARRAY(200) OF Soldier;
    PROCEDURE reorganize (my_unit IN OUT NOCOPY Platoon) IS ...
BEGIN
    ...
END;
```

For more information, see "Passing Large Data Structures with the NOCOPY Compiler Hint" on page 8-17.

Improving PL/SQL Performance with the RETURNING Clause

Often, applications need information about the row affected by a SQL operation, for example, to generate a report or take a subsequent action. The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to SELECT the row after an insert or update, or before a delete. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

In the following example, you update the salary of an employee and at the same time retrieve the employee’s name and new salary into PL/SQL variables.

```plsql
PROCEDURE update_salary (emp_id NUMBER) IS
    name    VARCHAR2(15);
    new_sal NUMBER;
BEGIN
```

```plsql
```
...  
UPDATE emp SET sal = sal * 1.1  
  WHERE empno = emp_id  
  RETURNING ename, sal INTO name, new_sal;  
-- Now do computations involving name and new_sal  
...  
END;  

Improving PL/SQL Performance with External Routines  

PL/SQL is specialized for SQL transaction processing. So, some tasks are more quickly done in a lower-level language such as C, which is very efficient at machine-precision calculations.

PL/SQL extends the functionality of the Oracle server by providing an interface for calling routines written in other languages. Standard libraries already written and available in other languages can be called from PL/SQL programs. This promotes reusability, efficiency, and modularity.

To speed up execution, you can rewrite computation-bound programs in C. In addition, you can move such programs from client to server, where they will execute faster thanks to more computing power and less across-network communication.

For example, you can write methods for an image object type in C, store them in a dynamic link library (DLL), register the library with PL/SQL, then call it from your applications. At run time, the library loads dynamically and, for safety, runs in a separate address space (implemented as a separate process).

For more information, see Oracle9i Application Developer’s Guide - Fundamentals.

Improving PL/SQL Performance with Object Types and Collections  

Collection types (see Chapter 5) and object types (see Chapter 10) increase your productivity by allowing for realistic data modeling. Complex real-world entities and relationships map directly into object types. And, a well-constructed object model can improve application performance by eliminating table joins, reducing round trips, and the like.

Client programs, including PL/SQL programs, can declare objects and collections, pass them as parameters, store them in the database, retrieve them, and so on. Also, by encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods.
Objects and collections are more efficient to store and retrieve because they can be manipulated as a whole. Also, object support is integrated with the database architecture, so it can take advantage of the many scalability and performance improvements built into each Oracle release.

Compiling PL/SQL Code for Native Execution

You can speed up PL/SQL procedures by compiling them into native code residing in shared libraries. The procedures are translated into C code, then compiled with your usual C compiler and linked into the Oracle process. You can use this technique with both the supplied Oracle packages, and procedures you write yourself. Because this technique cannot do much to speed up SQL statements called from PL/SQL, it is most effective for compute-intensive PL/SQL procedures that do not spend most of their time executing SQL.

To speed up one or more procedures using this technique:

1. Update the supplied makefile and enter the appropriate paths and other values for your system. The path of this makefile is $ORACLE_HOME/plsql/spnc_makefile.mk.

2. Use the ALTER SYSTEM or ALTER SESSION command, or update your initialization file, to set the parameter PLSQL_COMPILER_FLAGS to include the value NATIVE. The default setting includes the value INTERPRETED, and you must remove this keyword from the parameter value.

3. Compile one or more procedures, using one of these methods:
   - Use the ALTER PROCEDURE or ALTER PACKAGE command to recompile the procedure or the entire package.
   - Drop the procedure and create it again.
   - Use CREATE OR REPLACE to recompile the procedure.
   - Run one of the SQL*Plus scripts that sets up a set of Oracle-supplied packages.
   - Create a database using a preconfigured initialization file with PLSQL_COMPILER_FLAGS=NATIVE. During database creation, the UTLIRP script is run to compile all the Oracle-supplied packages.

4. To be sure that the process worked, you can query the data dictionary to see that a procedure is compiled for native execution. To check whether an existing procedure is compiled for native execution or not, you can query the data dictionary views USER_STORED_SETTINGS, DBA_STORED_SETTINGS, and...
ALL_STORED_SETTINGS. For example, to check the status of the procedure MY_PROC, you could enter:

SELECT param_value FROM user_stored_settings WHERE
  param_name = 'PLSQL_COMPILER_FLAGS'
  and object_name = 'MY_PROC';

The PARAM_VALUE column has a value of NATIVE for procedures that are compiled for native execution, and INTERPRETED otherwise.

After the procedures are compiled and turned into shared libraries, they are automatically linked into the Oracle process. You do not need to restart the database, or move the shared libraries to a different location. You can call back and forth between stored procedures, whether they are all compiled in the default way (interpreted), all compiled for native execution, or a mixture of both.

Because the PLSQL_COMPILER_FLAGS setting is stored inside the library unit for each procedure, procedures compiled for native execution are compiled the same way when the procedure is recompiled automatically after being invalidated, such as when a table that it depends on is recreated.

You can control the behavior of PL/SQL native compilation through the ALTER SYSTEM or ALTER SESSION commands or by setting or changing these parameters in the initialization file:

- PLSQL_COMPILER_FLAGS
- PLSQL_NATIVE_LIBRARY_DIR (cannot be set by ALTER SESSION for security reasons)
- PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT
- PLSQL_NATIVE_MAKEtility
- PLSQL_NATIVE_MAKE_FILE_NAME

See Also: Oracle9i Database Reference for complete details about the initialization parameters and data dictionary views.

Example of Compiling a PL/SQL Procedure for Native Execution

```sql
connect scott/tiger;
set serveroutput on;
alter session set plsql_native_library_dir='/home/orauser/lib';
alter session set plsql_native_make_utility='gmake';
alter session set plsql_native_make_file_name='/home/orauser/spnc_makefile.mk';
alter session set plsql_compiler_flags='NATIVE';
```
create or replace procedure hello_native_compilation
as
begin
    dbms_output.put_line('Hello world');
    select sysdate from dual;
end;

As the procedure is compiled, you see the various compilation and link commands being executed. The procedure is immediately available to call, and runs as a shared library directly within the Oracle process.

Limitations of Native Compilation

- If a package specification is compiled for native execution, the corresponding body should be compiled using the same setting.
- Debugging tools for PL/SQL do not handle procedures compiled for native execution.
- When many procedures and packages (typically, over 5000) are compiled for native execution, the large number of shared objects in a single directory might affect system performance. In this case, you can set the initialization parameter PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT in the initialization file before creating the database or compiling the PL/SQL packages or procedures. Set this parameter to a value (typically around 1000). Then create subdirectories underneath the directory specified in the PLSQL_NATIVE_LIBRARY_DIR parameter. Name the subdirectories d0, d1, d2 ... d999, up to the value specified for the subdirectory count. When the procedures are compiled for native execution, the DLLs are automatically distributed among these subdirectories by the PL/SQL compiler.
This chapter is a quick reference guide to PL/SQL syntax and semantics. It shows you how commands, parameters, and other language elements are sequenced to form PL/SQL statements. Also, to save you time and trouble, it provides usage notes and short examples.

This chapter discusses the following topics:

- Assignment Statement
- AUTONOMOUS TRANSACTION Pragma
- Blocks
- CASE Statement
- CLOSE Statement
- Collection Methods
- Collections
- Comments
- COMMIT Statement
- Constants and Variables
- Cursor Attributes
- Cursor Variables
- Cursors
- DELETE Statement
- EXCEPTION_INIT Pragma
- Exceptions
- EXECUTE IMMEDIATE Statement
- EXIT Statement
- Expressions
FETCH Statement
FORALL Statement
Functions
GOTO Statement
IF Statement
INSERT Statement
Literals
LOCK TABLE Statement
LOOP Statements
MERGE Statement
NULL Statement
Object Types
OPEN Statement
OPEN-FOR Statement
OPEN-FOR-USING Statement
Packages
Procedures
RAISE Statement
Records
RESTRICT_REFERENCES Pragma
RETURN Statement
ROLLBACK Statement
%ROWTYPE Attribute
SAVEPOINT Statement
SELECT INTO Statement
SERIALLY_REUSABLE Pragma
SET TRANSACTION Statement
SQL Cursor
SQLCODE Function
SQLERRM Function
%TYPE Attribute
UPDATE Statement
Reading the Syntax Diagrams

When you are unsure of the syntax to use in a PL/SQL statement, trace through its syntax diagram, reading from left to right and top to bottom. You can verify or construct any PL/SQL statement that way.

The diagrams are graphic representations of Bachus-Naur Form (BNF) productions. Within the diagrams, keywords are enclosed in boxes, delimiters in circles, and identifiers in ovals.

Each diagram defines a syntactic element. Every path through the diagram describes a possible form of that element. Follow in the direction of the arrows. If a line loops back on itself, you can repeat the element enclosed by the loop.
Assignment Statement

An assignment statement sets the current value of a variable, field, parameter, or element. The statement consists of an assignment target followed by the assignment operator and an expression. When the statement is executed, the expression is evaluated and the resulting value is stored in the target. For more information, see "Assignments" on page 2-22.

Syntax

```
assignment_statement
```

Keyword and Parameter Description

**attribute_name**

This identifies an attribute of an object type. The name must be unique within the object type (but can be reused in other object types). You cannot initialize an attribute in its declaration using the assignment operator or DEFAULT clause. Also, you cannot impose the NOT NULL constraint on an attribute.
**collection_name**
This identifies a nested table, index-by table, or varray previously declared within the current scope.

**cursor_variable_name**
This identifies a PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

**expression**
This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of expression, see "Expressions" on page 13-72. When the assignment statement is executed, the expression is evaluated and the resulting value is stored in the assignment target. The value and target must have compatible datatypes.

**field_name**
This identifies a field in a user-defined or %ROWTYPE record.

**host_cursor_variable_name**
This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

**host_variable_name**
This identifies a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Host variables must be prefixed with a colon.

**index**
This is a numeric expression that must yield a value of type BINARY_INTEGER or a value implicitly convertible to that datatype.
**indicator_name**
This identifies an indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable "indicates" the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables let you detect nulls or truncated values in output host variables.

**object_name**
This identifies an object (instance of an object type) previously declared within the current scope.

**parameter_name**
This identifies a formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears.

**record_name**
This identifies a user-defined or %ROWTYPE record previously declared within the current scope.

**variable_name**
This identifies a PL/SQL variable previously declared within the current scope.

**Usage Notes**
By default, unless a variable is initialized in its declaration, it is initialized to NULL every time a block or subprogram is entered. So, never reference a variable before you assign it a value.

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

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. When applied to an expression, the relational operators return a Boolean value. So, the following assignment is legal:

```sql
DECLARE
    out_of_range BOOLEAN;
    ...
BEGIN
    ...
    out_of_range := (salary < minimum) OR (salary > maximum);
```
As the next example shows, you can assign the value of an expression to a specific field in a record:

```
DECLARE
  emp_rec emp%ROWTYPE;
BEGIN
  ...
  emp_rec.sal := current_salary + increase;
END;
```

Moreover, you can assign values to all fields in a record at once. PL/SQL allows aggregate assignment between entire records if their declarations refer to the same cursor or table. For example, the following assignment is legal:

```
DECLARE
  emp_rec1 emp%ROWTYPE;
  emp_rec2 emp%ROWTYPE;
  dept_rec dept%ROWTYPE;
BEGIN
  ...
  emp_rec1 := emp_rec2;
END;
```

Using the following syntax, you can assign the value of an expression to a specific element in a collection:

```
collection_name(index) := expression;
```

In the following example, you assign the uppercase value of `last_name` to the third row in nested table `ename_tab`:

```
ename_tab(3) := UPPER(last_name);
```

Examples

Several examples of assignment statements follow:

```
wages := hours_worked * hourly_salary;
country := 'France';
costs := labor + supplies;
done := (count > 100);
department_rec.loc := 'BOSTON';
comm_tab(5) := sales * 0.15;
```

Related Topics

- Constants and Variables
- Expressions
- SELECT INTO Statement
The AUTONOMOUS_TRANSACTION pragma instructs the PL/SQL compiler to mark a routine as autonomous (independent). An autonomous transaction is an independent transaction started by another transaction, the main transaction. Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction. For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-53.

Syntax

```plaintext
autonomous_transaction pragma
```

Keyword and Parameter Description

**PRAGMA**

This keyword signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler.

Usage Notes

In this context, the term routine includes

- Top-level (not nested) anonymous PL/SQL blocks
- Local, standalone, and packaged functions and procedures
- Methods of a SQL object type
- Database triggers

You cannot 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. 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.
Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. So, you can log events, increment retry counters, and so on, even if the main transaction rolls back.

Unlike regular triggers, autonomous triggers can contain transaction control statements such as COMMIT and ROLLBACK. Also, unlike regular triggers, autonomous triggers can execute DDL statements (such as CREATE and DROP) using native dynamic SQL.

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. The changes also become visible to the main transaction when it resumes, but only if its isolation level is set to READ COMMITTED (the default).

If you set the isolation level of the main transaction to SERIALIZABLE, as follows, changes made by its autonomous transactions are not visible to the main transaction when it resumes:

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.

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. In that case, Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception. If the exception goes unhandled, the transaction is rolled back.

Examples

In the following example, you mark a packaged function as autonomous:

```
CREATE PACKAGE banking AS
  ...
  FUNCTION balance (acct_id INTEGER) RETURN REAL;
END banking;

CREATE PACKAGE BODY banking AS
  ...
  FUNCTION balance (acct_id INTEGER) RETURN REAL IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    my_bal REAL;
  BEGIN
```
... 
END;
END banking;

In the example below, you mark a database trigger as autonomous. Unlike regular triggers, autonomous triggers can contain transaction control statements.

CREATE TRIGGER parts_trigger
BEFORE INSERT ON parts FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO parts_log VALUES(:new.pnum, :new.pname);
  COMMIT;  -- allowed only in autonomous triggers
END;

Related Topics

EXCEPTION_INIT Pragma, RESTRICT_REFERENCES Pragma,
SERIALLY_RESUABLE Pragma
Blocks

The basic program unit in PL/SQL is the block. A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required. You can nest a block within another block wherever you can place an executable statement. For more information, see "Block Structure" on page 1-2 and "Scope and Visibility" on page 2-19.

Syntax
Keyword and Parameter Description

**base_type**
This is any scalar or user-defined PL/SQL datatype specifier such as CHAR, DATE, or RECORD.

**BEGIN**
This keyword signals the start of the executable part of a PL/SQL block, which contains executable statements. The executable part of a block is required. That is, a PL/SQL block must contain at least one executable statement. The **NULL** statement meets this requirement.
**collection_declaration**
This declares a collection (index-by table, nested table, or varray). For the syntax of collection_declaration, see “Collections” on page 13-28.

**constant_declaration**
This declares a constant. For the syntax of constant_declaration, see "Constants and Variables" on page 13-38.

**constraint**
This applies only to datatypes that can be constrained such as CHAR and NUMBER. For character datatypes, this specifies a maximum size in bytes. For numeric datatypes, this specifies a maximum precision and scale.

**cursor_declaration**
This declares an explicit cursor. For the syntax of cursor_declaration, see "Cursors" on page 13-53.

**cursor_variable_declaration**
This declares a cursor variable. For the syntax of cursor_variable_declaration, see "Cursor Variables" on page 13-47.

**DECLARE**
This keyword 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.

PL/SQL does not allow forward references. So, you must declare an item before referencing it in other statements, including other declarative statements. Also, you must declare subprograms at the end of a declarative section after all other program items.

**END**
This keyword signals the end of a PL/SQL block. It must be the last keyword in a block. Neither the END IF in an IF statement nor the END LOOP in a LOOP statement can substitute for the keyword END. Remember, END does not signal the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.
EXCEPTION
This keyword 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.

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, see Chapter 7.

exception_declaration
This declares an exception. For the syntax of exception_declaration, see "Exceptions" on page 13-63.

exception_handler
This associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of exception_handler, see "Exceptions" on page 13-63.

function_declaration
This declares a function. For the syntax of function_declaration, see "Functions" on page 13-90.

label_name
This is an undeclared identifier that optionally labels a PL/SQL block. If used, label_name must be enclosed by double angle brackets and must appear at the beginning of the block. Optionally, label_name (not enclosed by angle brackets) can also appear at the end of the block.

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, as the following example shows:

<<outer>>
DECLARE
  x INTEGER;
BEGIN
  ...

<<outer>>
DECLARE
  x INTEGER;
BEGIN
  ...

...
```plsql
DECLARE
  x INTEGER;
BEGIN
  ...
  IF x = outer.x THEN  -- refers to global x
    ...
  END IF;
  END;
END outer;
```

**object_declaration**
This declares an object (instance of an object type). For the syntax of `object_declaration`, see "Object Types" on page 13-120.

**procedure_declaration**
This declares a procedure. For the syntax of `procedure_declaration`, see "Procedures" on page 13-143.

**recordDeclaration**
This declares a user-defined record. For the syntax of `recordDeclaration`, see "Records" on page 13-150.

**statement**
This is an executable (not declarative) statement that you use to create algorithms. A sequence of statements can include procedural statements such as RAISE, SQL statements such as `UPDATE`, and PL/SQL blocks (sometimes called "block statements").

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**
This identifies a user-defined subtype that was defined using any scalar or user-defined PL/SQL datatype specifier such as `CHAR`, `DATE`, or `RECORD`.

**variable_declaration**
This declares a variable. For the syntax of `variable_declaration`, see "Constants and Variables" on page 13-38.
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.

Example

The following PL/SQL block declares several variables and constants, then calculates a ratio using values selected from a database table:

```sql
-- available online in file 'examp11'
DECLARE
    numerator   NUMBER;
    denominator NUMBER;
    the_ratio   NUMBER;
    lower_limit CONSTANT NUMBER := 0.72;
    samp_num    CONSTANT NUMBER := 132;
BEGIN
    SELECT x, y INTO numerator, denominator FROM result_table
        WHERE sample_id = samp_num;
    the_ratio := numerator/denominator;
    IF the_ratio > lower_limit THEN
        INSERT INTO ratio VALUES (samp_num, the_ratio);
    ELSE
        INSERT INTO ratio VALUES (samp_num, -1);
    END IF;
    COMMIT;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        INSERT INTO ratio VALUES (samp_num, 0);
    COMMIT;
    WHEN OTHERS THEN
        ROLLBACK;
END;
```

Related Topics

Constants and Variables, Exceptions, Functions, Procedures
The `CASE` statement selects a sequence of statements to execute. To select the sequence, the `CASE` statement uses a selector (an expression whose value is used to select one of several alternatives) or, in the searched `CASE` statement, multiple search conditions.

See also:.

**Syntax**

```sql
searched_case_statement ::=  
    [ <<label_name>> ]  
    CASE { WHEN boolean_expression THEN {statement; } ... }...  
    [ ELSE {statement; }... ]  
    END CASE [ label_name ];

simple_case_statement ::=  
    [ <<label_name>> ]  
    CASE case_operand  
    { WHEN when_operand THEN {statement; } ... }...  
    [ ELSE {statement; }... ]  
    END CASE [ label_name ];
```

**Keyword and Parameter Description**

The value of the `CASE` operand and `WHEN` operands in a simple `CASE` statement can be any PL/SQL type other than `BLOB`, `BFILE`, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

If the `ELSE` clause is omitted, the system substitutes a default action. For a `CASE` statement, the default when none of the conditions matches is to raise a `CASE_NOT_FOUND` exception. For a `CASE` expression, the default is to return `NULL`.
Usage Notes

Each WHEN clause is executed only once.

The WHEN clauses are executed in order.

After a matching WHEN clause is found, subsequent WHEN clauses are not executed.

Because the execution order of the WHEN clauses is defined by the preceding rules, the statements in a WHEN clause are allowed to modify the database and call non-deterministic functions.

There is no "fall-through" as in the C switch statement. Once a WHEN clause is matched and its statements are executed, the CASE statement ends.

The CASE statement is appropriate when there is some different action to be taken for each alternative. If you just need to choose among several values to assign to a variable, you can code an assignment statement using a CASE expression instead.

Examples

The following example shows a simple CASE statement. Notice that you can use multiple statements after a WHEN clause.

```
DECLARE
  n number;
BEGIN
  CASE n
    WHEN 1 THEN dbms_output.put_line('n = 1');
    WHEN 2 THEN
      dbms_output.put_line('n = 2');
      dbms_output.put_line('That implies n > 1');
    ELSE dbms_output.put_line('n is some other value.');
  END CASE;
END;
```
The following example shows a searched CASE statement. Notice that the WHEN clauses can use different conditions rather than all testing the same variable or using the same operator. Because this example does not use an ELSE clause, an exception is raised if none of the WHEN conditions are met.

DECLARE
    quantity NUMBER;
    projected NUMBER;
    needed NUMBER;
BEGIN
    <<here>>
    CASE
        WHEN quantity is null THEN
            dbms_output.put_line('Quantity not available');
        WHEN quantity + projected >= needed THEN
            dbms_output.put_line('Quantity ' || quantity || ' should be enough if projections are met.);
        WHEN quantity >= 0 THEN
            dbms_output.put_line('Quantity ' || quantity || ' is probably not enough.');</n
    END CASE here;
EXCEPTION
    WHEN CASE_NOT_FOUND THEN
        dbms_output.put_line('Somehow quantity must be less than 0.');
END;

Related Topics

"Conditional Control: IF and CASE Statements" on page 4-2, "CASE Expressions" on page 2-31, NULLIF and COALESCE expressions in Oracle9i SQL Reference
CLOSE Statement

The **CLOSE** statement allows resources held by an open cursor or cursor variable to be reused. No more rows can be fetched from a closed cursor or cursor variable. For more information, see "Managing Cursors" on page 6-6.

**Syntax**

```
close_statement
```

**Keyword and Parameter Description**

- **cursor_name**
  This identifies an explicit cursor previously declared within the current scope and currently open.

- **cursor_variable_name**
  This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope and currently open.

- **host_cursor_variable_name**
  This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

**Usage Notes**

Once a cursor or cursor variable is closed, you can reopen it using the **OPEN** or **OPEN-FOR** statement, respectively. If you reopen a cursor without closing it first, PL/SQL raises the predefined exception **CURSOR_ALREADY_OPEN**. However, you need not close a cursor variable before reopening it.
If you try to close an already-closed or never-opened cursor or cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Example

In the following example, after the last row is fetched and processed, you close the cursor variable emp_cv:

```plsql
LOOP
    FETCH emp_cv INTO emp_rec;
    EXIT WHEN emp_cv%NOTFOUND;
    ... -- process data record
END LOOP;
/* Close cursor variable. */
CLOSE emp_cv;
```

Related Topics

FETCH Statement, OPEN Statement, OPEN-FOR Statement
A collection method is a built-in function or procedure that operates on collections and is called using dot notation. The methods EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, NEXT, EXTEND, TRIM, and DELETE help generalize code, make collections easier to use, and make your applications easier to maintain.

EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, and NEXT are functions, which appear as part of an expression. EXTEND, TRIM, and DELETE are procedures, which appear as a statement. EXISTS, PRIOR, NEXT, TRIM, EXTEND, and DELETE take integer parameters. EXTEND and TRIM cannot be used with index-by tables.

For more information, see "Using Collection Methods" on page 5-23.

Syntax

collection_method_call

---

Collection Methods
Keyword and Parameter Description

**collection_name**
This identifies an index-by table, nested table, or varray previously declared within the current scope.

**COUNT**
COUNT returns the number of elements that a collection currently contains, which is useful because the current size of a collection is not always known. You can use COUNT wherever an integer expression is allowed.

For varrays, COUNT always equals LAST. For nested tables, normally, COUNT equals LAST. But, if you delete elements from the middle of a nested table, COUNT is smaller than LAST.

**DELETE**
This procedure has three forms. DELETE removes all elements from a collection.
DELETE (n) removes the n\(^{th}\) element from an index-by table or nested table. If n is null, DELETE (n) does nothing. DELETE (m, n) removes all elements in the range m..n from an index-by table or nested table. If m is larger than n or if m or n is null, DELETE (m, n) does nothing.

**EXISTS**
EXISTS (n) returns TRUE if the n\(^{th}\) element in a collection exists. Otherwise, EXISTS (n) returns FALSE. Mainly, you use EXISTS with DELETE to maintain sparse nested tables. You can also use EXISTS to avoid raising an exception when you reference a nonexistent element. When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

**EXTEND**
This procedure has three forms. EXTEND appends one null element to a collection.
EXTEND (n) appends n null elements to a collection. EXTEND (n, i) appends n copies of the i\(^{th}\) element to a collection. EXTEND operates on the internal size of a collection. So, if EXTEND encounters deleted elements, it includes them in its tally. You cannot use EXTEND with index-by tables.
**FIRST, LAST**

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection. If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same index number.

For varrays, FIRST always returns 1 and LAST always equals COUNT. For nested tables, normally, LAST equals COUNT. But, if you delete elements from the middle of a nested table, LAST is larger than COUNT.

**index**

This is an expression that must yield (or convert implicitly to) an integer.

**LIMIT**

For nested tables, which have no maximum size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain (which you must specify in its type definition).

**NEXT, PRIOR**

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. Likewise, if n has no successor, NEXT (n) returns NULL.

**TRIM**

This procedure has two forms. TRIM removes one element from the end of a collection. TRIM (n) removes n elements from the end of a collection. If n is greater than COUNT, TRIM (n) raises SUBSCRIPT_BEYOND_COUNT. You cannot use TRIM with index-by tables.

TRIM operates on the internal size of a collection. So, if TRIM encounters deleted elements, it includes them in its tally.

**Usage Notes**

You cannot use collection methods in a SQL statement. If you try, you get a compilation error.

Only EXISTS can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises COLLECTION_IS_NULL.

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 have been deleted.
EXTEND operates on the internal size of a collection, which includes deleted elements. You cannot use EXTEND to initialize an atomically null collection. Also, if you impose the NOT NULL constraint on a TABLE or VARRAY type, you cannot apply the first two forms of EXTEND to collections of that type.

If an element to be deleted does not exist, DELETE simply skips it; no exception is raised. Varrays are dense, so you cannot delete their individual elements.

PL/SQL keeps placeholders for deleted elements. So, you can replace a deleted element simply by assigning it a new value. However, PL/SQL does not keep placeholders for trimmed elements.

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

In general, do not depend on the interaction between TRIM and DELETE. It is better to treat nested tables like fixed-size arrays and use only DELETE, or to treat them like stacks and use only TRIM and EXTEND.

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. So, 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

In the following example, you use NEXT to traverse a nested table from which some elements have been deleted:

```plsql
i := courses.FIRST;  -- get subscript of first element
WHILE i IS NOT NULL LOOP
   -- do something with courses(i)
   i := courses.NEXT(i);  -- get subscript of next element
END LOOP;
```

In the following example, PL/SQL executes the assignment statement only if element i exists:

```plsql
IF courses.EXISTS(i) THEN
   courses(i) := new_course;
END IF;
```

The next example shows that you can use FIRST and LAST to specify the lower and upper bounds of a loop range provided each element in that range exists:

```plsql
FOR i IN courses.FIRST..courses.LAST LOOP ...
```
In the following example, you delete elements 2 through 5 from a nested table:

```plsql
courses.DELETE(2, 5);
```

In the final example, you use LIMIT to determine if you can add 20 more elements to varray `projects`:

```plsql
IF (projects.COUNT + 20) < projects.LIMIT THEN
  -- add 20 more elements
```

**Related Topics**

Collections, Functions, Procedures
A collection is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers three kinds of collections: index-by tables, nested tables, and varrays (short for variable-size arrays). Nested tables extend the functionality of index-by tables (formerly called "PL/SQL tables").

Collections work like the arrays found in most third-generation programming languages. Collections can have only one dimension and must be indexed by integers. In some languages, such as Ada and Pascal, arrays can have multiple dimensions and can be indexed by enumeration types. To model multi-dimensional arrays, you can declare collections whose items are other collections.

Nested tables and varrays can store instances of an object type and, conversely, can be attributes of an object type. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

For more information, see "Defining and Declaring Collections" on page 5-5.
Syntax

**table_type_definition**

```
TYPE type_name IS TABLE OF element_type
INDEX BY BINARY_INTEGER;
```

**varray_type_definition**

```
TYPE type_name IS
  VARRAY VARRying ARRAY(size_limit)
  OF element_type
  NOT NULL;
```

**collection_declaration**

```
collection_name type_name;
```
Keyword and Parameter Description

**element_type**

This is any PL/SQL datatype except **BINARY_INTEGER**, **BOOLEAN**, **LONG**, **LONG RAW**, **NATURAL**, **NATURALN**, **NCHAR**, **NCLOB**, **NVARCHAR2**, **PLS_INTEGER**, **POSITIVE**, **POSITIVEN**, **REF CURSOR**, **SIGNTYPE**, or **STRING**. Also, with varrays, **element_type** cannot be **BLOB**, **CLOB**, or an object type with **BLOB** or **CLOB** attributes.

**INDEX BY BINARY_INTEGER**

This optional clause lets you define Version 2 PL/SQL tables, which are now called index-by tables.

**size_limit**

This is a positive integer literal that specifies the maximum size of a varray, which is the maximum number of elements the varray can contain.
**type_name**
This identifies a user-defined collection type that was defined using the datatype specifier `TABLE` or `VARRAY`.

**Usage Notes**

Nested tables extend the functionality of index-by tables, so they differ in several ways. See "Nested Tables Versus Index-By Tables" on page 5-3.

Every element reference includes the collection name and a subscript enclosed in parentheses; the subscript determines which element is processed. Except for index-by tables, which can have negative subscripts, collection subscripts have a fixed lower bound of 1.

You can define all three collection types in the declarative part of any PL/SQL block, subprogram, or package. But, only nested table and varray types can be created and stored in an Oracle database.

Index-by tables and nested tables can be sparse (have non-consecutive subscripts), but varrays are always dense (have consecutive subscripts). Unlike nested tables, varrays retain their ordering and subscripts when stored in the database.

Initially, index-by tables are sparse. That enables you, for example, to store reference data in a temporary index-by table using a numeric 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" a collection from the elements passed to it.

Because nested tables and varrays can be atomically null, they can be tested for nullity. However, they cannot be compared for equality or inequality. This restriction also applies to implicit comparisons. For example, collections cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` list.
Collections can store instances of an object type and, conversely, can be attributes of an object type. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

When calling a function that returns a collection, you use the following syntax to reference elements in the collection:

```sql
collection_name(parameter_list)(subscript)
```

With the Oracle Call Interface (OCI) or the Oracle Precompilers, you can bind host arrays to index-by tables declared as the formal parameters of a subprogram. That lets you pass host arrays to stored functions and procedures.

**Examples**

To specify the element type of a collection, you can use `%TYPE` or `%ROWTYPE`, as the following example shows:

```sql
DECLARE
    TYPE JobList IS VARRAY(10) OF emp.job%TYPE;  -- based on column
    CURSOR c1 IS SELECT * FROM dept;
    TYPE DeptFile IS TABLE OF c1%ROWTYPE;  -- based on cursor
    TYPE EmpFile IS VARRAY(150) OF emp%ROWTYPE; -- based on database table

    IN the next example, you use a RECORD type to specify the element type:

    DECLARE
        TYPE Entry IS RECORD (
            term    VARCHAR2(20),
            meaning VARCHAR2(200));
        TYPE Glossary IS VARRAY(250) OF Entry;

    In the example below, you declare an index-by table of records. Each element of the table stores a row from the emp database table.

    DECLARE
        TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
            INDEX BY BINARY_INTEGER;
        emp_tab EmpTabTyp;
    BEGIN
        /* Retrieve employee record. */
        SELECT * INTO emp_tab(7468) FROM emp WHERE empno = 7468;
```
When defining a VARRAY type, you must specify its maximum size. In the following example, you define a type that stores up to 366 dates:

```
DECLARE
    TYPE Calendar IS VARRAY(366) OF DATE;
```

Once you define a collection type, you can declare collections of that type, as the following SQL*Plus script shows:

```
CREATE TYPE Project AS OBJECT(
    project_no NUMBER(2),
    title      VARCHAR2(35),
    cost       NUMBER(7,2))
/
CREATE TYPE ProjectList AS VARRAY(50) OF Project  -- VARRAY type
/
CREATE TABLE department (
    idnum    NUMBER(2),
    name     VARCHAR2(15),
    budget   NUMBER(11,2),
    projects ProjectList)  -- declare varray
/
```

The identifier `projects` represents an entire varray. Each element of `projects` will store a `Project` object.

In the following example, you declare a nested table as the formal parameter of a packaged procedure:

```
CREATE PACKAGE personnel AS
    TYPE Staff IS TABLE OF Employee;
    ...
    PROCEDURE award_bonuses (members IN Staff);
```

You can specify a collection type in the RETURN clause of a function spec, as the following example shows:

```
DECLARE
    TYPE SalesForce IS VARRAY(20) OF Salesperson;
    FUNCTION top_performers (n INTEGER) RETURN SalesForce IS ...
```
In the following example, you update the list of projects assigned to the Security Department:

```plaintext
DECLARE
    new_projects ProjectList :=
        ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                     Project(2, 'Inspect Emergency Exits', 1900),
                     Project(3, 'Upgrade Alarm System', 3350),
                     Project(4, 'Analyze Local Crime Stats', 825));
BEGIN
    UPDATE department
    SET projects = new_projects WHERE name = 'Security';
END;
```

In the next example, you retrieve all the projects for the Accounting Department into a local varray:

```plaintext
DECLARE
    my_projects ProjectList;
BEGIN
    SELECT projects INTO my_projects FROM department
    WHERE name = 'Accounting';
END;
```

**Related Topics**

Collection Methods, Object Types, Records
Comments

Comments describe the purpose and use of code segments and so promote readability. PL/SQL supports two comment styles: single-line and multi-line. Single-line comments begin with a double hyphen (--) anywhere on a line and extend to the end of the line. Multi-line comments begin with a slash-asterisk (/*), end with an asterisk-slash (*/), and can span multiple lines. For more information, see “Comments” on page 2-10.

Syntax

comment

Usage Notes

Comments can appear within a statement at the end of a line. However, you cannot nest comments.

You cannot use single-line comments in a PL/SQL block that will be processed dynamically by an Oracle Precompiler program because end-of-line characters are ignored. As a result, single-line comments extend to the end of the block, not just to the end of a line. Instead, use multi-line comments.

While testing or debugging a program, you might want to disable a line of code. The following example shows how you can "comment-out" the line:

-- UPDATE dept SET loc = my_loc WHERE deptno = my_deptno;

You can use multi-line comment delimiters to comment-out whole sections of code.

Examples

The following examples show various comment styles:

-- compute the area of a circle
area := pi * radius**2; -- pi equals 3.14159
/* Compute the area
   of a circle. */
area := pi * radius**2; /* pi equals 3.14159 */
The COMMIT statement explicitly makes permanent any changes made to the database during the current transaction. Changes made to the database are not considered permanent until they are committed. A commit also makes the changes visible to other users. For more information, see "Overview of Transaction Processing in PL/SQL" on page 6-43.

**Syntax**

```
commit_statement

COMMlIT WORK COMMENT text
```

**Keyword and Parameter Description**

- **COMMENT**
  This keyword specifies a comment to be associated with the current transaction and is typically used with distributed transactions. The text must be a quoted literal no more than 50 characters long.

- **WORK**
  This keyword is optional and has no effect except to improve readability.

**Usage Notes**

The COMMIT statement releases all row and table locks. It also erases any savepoints you marked since the last commit or rollback. Until your changes are committed, the following conditions hold:

- You can see the changes when you query the tables you modified, but other users cannot see the changes.

- If you change your mind or need to correct a mistake, you can use the ROLLBACK statement to roll back (undo) the changes.
If you commit while a `FOR UPDATE` cursor is open, a subsequent fetch on that cursor raises an exception. The cursor remains open, however, so you should close it. For more information, see "Using FOR UPDATE" on page 6-50.

When a distributed transaction fails, the text specified by `COMMENT` helps you diagnose the problem. If a distributed transaction is ever in doubt, Oracle stores the text in the data dictionary along with the transaction ID. For more information about distributed transactions, see *Oracle9i Database Concepts*.

In SQL, the `FORCE` clause manually commits an in-doubt distributed transaction. However, PL/SQL does not support this clause. For example, the following statement is illegal:

```
COMMIT WORK FORCE '23.51.54';  -- illegal
```

In embedded SQL, the `RELEASE` option frees all Oracle resources (locks and cursors) held by a program and disconnects from the database. However, PL/SQL does not support this option. For example, the following statement is illegal:

```
COMMIT WORK RELEASE;  -- illegal
```

**Related Topics**

ROLLBACK Statement, SAVEPOINT Statement
You can declare constants and variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the `NOT NULL` constraint. For more information, see "Declarations" on page 2-11.

**Syntax**

```
variable_name datatype
```

```
datatype
```

```
  variable_name
  collection_name % TYPE
  collection_type_name
  cursor_name % ROWTYPE
  cursor_variable_name % TYPE
  db_table_name % ROWTYPE
  column_name % TYPE
  object_name % TYPE
  REF
  object_type_name
  record_name % TYPE
  record_type_name
  ref_cursor_type_name
  scalar_datatype_name
  variable_name % TYPE
```
Keyword and Parameter Description

collection_name
Identifies a collection (index-by table, nested table, or varray) previously declared within the current scope.

collection_type_name
Identifies a user-defined collection type that was defined using the datatype specifier TABLE or VARRAY.

CONSTANT
Denotes the declaration of a constant. You must initialize a constant in its declaration. Once initialized, the value of a constant cannot be changed.

constant_name
Identifies a program constant. For naming conventions, see "Identifiers" on page 2-4.

cursor_name
Identifies an explicit cursor previously declared within the current scope.

cursor_variable_name
Identifies 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**
An arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of expression is assigned to the constant or variable. The value and the constant or variable must have compatible datatypes.

**NOT NULL**
This constraint prevents the assigning of nulls to a variable or constant. At run time, trying to assign a null to a variable defined as NOT NULL raises the predefined exception VALUE_ERROR. The constraint NOT NULL must be followed by an initialization clause.

**object_name**
Identifies an instance of an object type previously declared within the current scope.

**record_name**
Identifies a user-defined or %ROWTYPE record previously declared within the current scope.

**record_name.field_name**
Identifies a field in a user-defined or %ROWTYPE record previously declared within the current scope.

**record_type_name**
Identifies a user-defined record type that is defined using the datatype specifier RECORD.

**ref_cursor_type_name**
Identifies a user-defined cursor variable type that was defined using the datatype specifier REF CURSOR.

**%ROWTYPE**
This attribute provides a record type that represents a row in a database table or a row fetched from a previously declared cursor. Fields in the record and corresponding columns in the row have the same names and datatypes.
**scalar_datatype_name**
Identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2. Includes any qualifiers for size, precision, or character versus byte semantics.

**%TYPE**
This attribute provides the datatype of a previously declared collection, cursor variable, field, object, record, database column, or variable.

**variable_name**
Identifies a program variable.

**Usage Notes**
Constants and variables are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL. So, unless you expressly initialize a variable, its value is undefined.

Whether public or private, constants and variables declared in a package spec are initialized only once for each session.

An initialization clause is required when declaring NOT NULL variables and when declaring constants. If you use %ROWTYPE to declare a variable, initialization is not allowed.

**Examples**
Several examples of variable and constant declarations follow:

```sql
credit_limit CONSTANT NUMBER := 5000;
invalid BOOLEAN := FALSE;
acct_id INTEGER(4) NOT NULL DEFAULT 9999;
pi CONSTANT REAL := 3.14159;
postal_code VARCHAR2(20);
last_name VARCHAR2(20 CHAR);
my_ename emp.ename%TYPE;
```

**Related Topics**
"Declarations" on page 2-11, "Predefined Datatypes" on page 3-2, Assignment Statement, Expressions, %ROWTYPE Attribute, %TYPE Attribute
Cursor Attributes

Every explicit cursor and cursor variable has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a data manipulation statement. For more information, see "Using Cursor Attributes" on page 6-34.

The implicit cursor SQL has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS. For more information, see "SQL Cursor" on page 13-173.

Syntax

cursor_attribute

cursor_name

cursor_variable_name

host_cursor_variable_name

%

FOUND

ISOPEN

NOTFOUND

ROWCOUNT

Keyword and Parameter Description

cursor_name
This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name
This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope.

%FOUND Attribute
This is 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 yields NULL. Thereafter, it yields TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row.
host_cursor_variable_name
This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

%ISOPEN Attribute
This is a cursor attribute that can be appended to the name of a cursor or cursor variable. If a cursor is open, cursor_name%ISOPEN yields TRUE; otherwise, it yields FALSE.

%NOTFOUND Attribute
This is 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 yields NULL. Thereafter, it yields FALSE if the last fetch returned a row, or TRUE if the last fetch failed to return a row.

%ROWCOUNT Attribute
This is 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 yields 0. Thereafter, it yields 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. So, 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 have been 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. So, if FETCH never executes successfully, the loop is never exited. That is 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;
```

You can use the cursor attributes in procedural statements but not in SQL statements.

### Examples

The PL/SQL block below uses %FOUND to select an action. The IF statement either inserts a row or exits the loop unconditionally.

```plsql
-- available online in file 'examp12'
DECLARE
    CURSOR num1_cur IS SELECT num FROM num1_tab
        ORDER BY sequence;
    CURSOR num2_cur IS SELECT num FROM num2_tab
        ORDER BY sequence;
    num1     num1_tab.num%TYPE;
    num2     num2_tab.num%TYPE;
    pair_num NUMBER := 0;
BEGIN
    OPEN num1_cur;
    OPEN num2_cur;
    LOOP   -- loop through the two tables and get pairs of numbers
        FETCH num1_cur INTO num1;
        FETCH num2_cur INTO num2;
        IF (num1_cur%FOUND) AND (num2_cur%FOUND) THEN
            pair_num := pair_num + 1;
            INSERT INTO sum_tab VALUES (pair_num, num1 + num2);
        ELSE
            EXIT;
        END IF;
    END LOOP;
    CLOSE num1_cur;
    CLOSE num2_cur;
END;
```
The next example uses the same block. However, instead of using `%FOUND` in an `IF` statement, it uses `%NOTFOUND` in an `EXIT WHEN` statement.

```
-- available online in file 'examp13'
DECLARE
  CURSOR num1_cur IS SELECT num FROM num1_tab
    ORDER BY sequence;
  CURSOR num2_cur IS SELECT num FROM num2_tab
    ORDER BY sequence;
  num1   num1_tab.num%TYPE;
  num2   num2_tab.num%TYPE;
  pair_num NUMBER := 0;
BEGIN
  OPEN num1_cur;
  OPEN num2_cur;
  LOOP   -- loop through the two tables and get
      -- pairs of numbers
      FETCH num1_cur INTO num1;
      FETCH num2_cur INTO num2;
      EXIT WHEN (num1_cur%NOTFOUND) OR (num2_cur%NOTFOUND);
      pair_num := pair_num + 1;
      INSERT INTO sum_tab VALUES (pair_num, num1 + num2);
    END LOOP;
  CLOSE num1_cur;
  CLOSE num2_cur;
END;
```

In the following example, you use `%ISOPEN` to make a decision:

```
IF NOT (emp_cur%ISOPEN) THEN
  OPEN emp_cur;
END IF;
FETCH emp_cur INTO emp_rec;
```

The following PL/SQL block uses `%ROWCOUNT` to fetch the names and salaries of the five highest-paid employees:

```
-- available online in file 'examp14'
DECLARE
  CURSOR c1 is
    SELECT ename, empno, sal FROM emp
      ORDER BY sal DESC;   -- start with highest-paid employee
  my_ename CHAR(10);
  my_empno NUMBER(4);
  my_sal   NUMBER(7,2);
```
Cursor Attributes

BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_empn, my_sal;
    EXIT WHEN (c1%ROWCOUNT > 5) OR (c1%NOTFOUND);
    INSERT INTO temp VALUES (my_sal, my_empn, my_ename);
    COMMIT;
  END LOOP;
  CLOSE c1;
END;

In the final example, you use %ROWCOUNT to raise an exception if an unexpectedly high number of rows is deleted:

DELETE FROM accts WHERE status = 'BAD DEBT';
IF SQL%ROWCOUNT > 10 THEN
  RAISE out_of_bounds;
END IF;

Related Topics

Cursors, Cursor Variables
Cursor Variables

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use an explicit cursor, which names the work area. Or, you can use a cursor variable, which points to the work area. Whereas a cursor always refers to the same query work area, a cursor variable can refer to different work areas. To create cursor variables, you define a REF CURSOR type, then declare cursor variables of that type.

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, not an item.

For more information, see "Using Cursor Variables" on page 6-16.

Syntax

```
TYPE type_name IS REF CURSOR;
cursor_variable_name type_name;
RETURN db_table_name cursor_name cursor_variable_name % ROWTYPE;
RETURN record_name % TYPE;
RETURN record_type_name rel_cursor_type_name;
```

```
cursor_variable_declaration
cursor_variable_name type_name;
```
Keyword and Parameter Description

**cursor_name**
This identifies an explicit cursor previously declared within the current scope.

**cursor_variable_name**
This identifies a PL/SQL cursor variable previously declared within the current scope.

**db_table_name**
This identifies a database table (or view) that must be accessible when the declaration is elaborated.

**record_name**
This identifies a user-defined record previously declared within the current scope.

**record_type_name**
This identifies a user-defined record type that was defined using the datatype specifier `RECORD`.

**REF CURSOR**
In PL/SQL, pointers have datatype `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects. Therefore, cursor variables have datatype `REF CURSOR`.

**RETURN**
This keyword introduces the `RETURN` clause, which specifies the datatype of a cursor variable return value. You can use the `%ROWTYPE` attribute in the `RETURN` clause to provide a record type that represents a row in a database table or a row returned by a cursor or strongly typed cursor variable. Also, you can use the `%TYPE` attribute to provide the datatype of a previously declared record.

**%ROWTYPE**
This attribute provides a record type that represents a row in a database table or a row fetched from a cursor or strongly typed cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.
Cursor Variables

%TYPE
This attribute provides the datatype of a previously declared user-defined record.

type_name
This is a user-defined cursor variable type that was defined using the datatype specifier REF CURSOR.

Usage Notes
Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as a bind variable to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side.

The Oracle database server also has a PL/SQL engine. So, you can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs). And, 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.

Mainly, you use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle server can all refer to the same work area.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). 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 the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Once you define a REF CURSOR type, you can declare cursor variables of that type. You can use %TYPE to provide the datatype of a record variable. Also, in the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable.

Currently, cursor variables are subject to several restrictions. See ”Restrictions on Cursor Variables” on page 6-33.
You use three statements to control a cursor variable: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set. When all the rows are processed, you CLOSE the cursor variable.

Other OPEN-FOR statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. When you reopen a cursor variable for a different query, the previous query is lost.

PL/SQL makes sure the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible field or variable in the INTO clause. Also, the number of fields or variables must equal the number of column values. Otherwise, you get an error.

If both cursor variables involved in an assignment are strongly typed, they must have the same datatype. However, if one or both cursor variables are weakly typed, they need not have the same datatype.

When declaring a cursor variable as the formal parameter of a subprogram that fetches from or closes the cursor variable, you must specify the IN or IN OUT mode. If the subprogram opens the cursor variable, you must specify the IN OUT mode.

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises ROWTYPE_MISMATCH if the return types of the actual and formal parameters are incompatible.

You can apply the cursor attributes %FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT to a cursor variable.

If you try to fetch from, close, or apply cursor attributes to a cursor variable that does not point to a query work area, PL/SQL raises the predefined exception INVALID_CURSOR. You can make a cursor variable (or parameter) point to a query work area in two ways:

- OPEN the cursor variable FOR the query.
- Assign to the cursor variable the value of an already OPENed host cursor variable or PL/SQL cursor variable.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.
Examples

You can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program. To use the host cursor variable, you must pass it as a bind variable to PL/SQL. In the following Pro*C example, you pass a host cursor variable and a selector to a PL/SQL block, which opens the cursor variable for the chosen query:

```sql
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 emp;
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM dept;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM salgrade;
  END IF;
END;
END-EXEC;
```

Host cursor variables are compatible with any query return type. They behave just like weakly typed PL/SQL cursor variables.

When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping `OPEN-FOR` statements. For example, the following PL/SQL block opens three cursor variables in a single round-trip:

```sql
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM emp;
  OPEN :dept_cv FOR SELECT * FROM dept;
  OPEN :grade_cv FOR SELECT * FROM salgrade;
END;
```
You can also pass a cursor variable to PL/SQL by calling a stored procedure that declares a cursor variable as one of its formal parameters. To centralize data retrieval, you can group type-compatible queries in a packaged procedure, as the following example shows:

```sql
CREATE PACKAGE emp_data AS
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice IN NUMBER);
END emp_data;

CREATE PACKAGE BODY emp_data AS
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice IN NUMBER) IS
        BEGIN
            IF choice = 1 THEN
                OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
            ELSIF choice = 2 THEN
                OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
            ELSIF choice = 3 THEN
                OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
            END IF;
        END open_emp_cv;
END emp_data;
```

Alternatively, you can use a standalone procedure to open the cursor variable. Simply define the REF CURSOR type in a separate package, then reference that type in the standalone procedure. For instance, if you create the following (bodiless) package, you can create standalone procedures that reference the types it defines:

```sql
CREATE PACKAGE cv_types AS
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
    TYPE BonusCurTyp IS REF CURSOR RETURN bonus%ROWTYPE;
    ...
END cv_types;
```

**Related Topics**

CLOSE Statement, Cursor Attributes, Cursors, FETCH Statement, OPEN-FOR Statement
Cursors

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. An explicit cursor lets you name the work area, access the information, and process the rows individually. For more information, see "Managing Cursors" on page 6-6.

Syntax

cursor_declaration

CURSOR cursor_name (cursor_parameter_declaration)

RETURN rowtype IS

select_statement ;

cursor_spec

CURSOR cursor_name (cursor_parameter_declaration)

RETURN rowtype ;

cursor_body

CURSOR cursor_name (cursor_parameter_declaration)

RETURN rowtype IS

select_statement ;

cursor_parameter_declaration

parameter_name IN datatype DEFAULT expression
Keyword and Parameter Description

**cursor_name**
This identifies an explicit cursor previously declared within the current scope.

**datatype**
This is a type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 13-38.

**db_table_name**
This identifies a database table (or view) that must be accessible when the declaration is elaborated.

**expression**
This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

**parameter_name**
This identifies a cursor parameter; that is, 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
This identifies a user-defined record previously declared within the current scope.

record_type_name
This identifies a user-defined record type that was defined using the datatype specifier RECORD.

RETURN
This keyword introduces the RETURN clause, which specifies the datatype of a cursor return value. You can use the %ROWTYPE attribute in the RETURN clause to provide a record type that represents a row in a database table or a row returned by a previously declared cursor. Also, you can use the %TYPE attribute to provide the datatype of a previously declared record.

A cursor body must have a SELECT statement and the same RETURN clause as its corresponding cursor spec. Also, the number, order, and datatypes of select items in the SELECT clause must match the RETURN clause.

%ROWTYPE
This attribute provides a record type that represents a row in a database table or a row fetched from a previously declared cursor or cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.

select_statement
This is a query that returns a result set of rows. Its syntax is like that of select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 13-164. If the cursor declaration declares parameters, each parameter must be used in the query.

%TYPE
This attribute provides the datatype of a previously declared user-defined record.

Usage Notes
You must declare a cursor before referencing it in an OPEN, FETCH, or CLOSE statement. And, you must declare a variable before referencing it in a cursor declaration. The word SQL is reserved by PL/SQL for use as the default name for implicit cursors and cannot be used in a cursor declaration.
You cannot assign values to a cursor name or use it in an expression. However, cursors and variables follow the same scoping rules. For more information, see "Scope and Visibility" on page 2-19.

You retrieve data from a cursor by opening it, then fetching from it. Because the FETCH statement specifies the target variables, using an INTO clause in the SELECT statement of a cursor declaration is redundant and invalid.

The scope of cursor parameters is local to the cursor, meaning that they can be referenced only within the query used in the cursor declaration. The values of cursor parameters are used by the associated query when the cursor is opened. The query can also reference other PL/SQL variables within its scope.

The datatype of a cursor parameter must be specified without constraints. For example, the following parameter declarations are illegal:

```
CURSOR c1 (emp_id NUMBER NOT NULL, dept_no NUMBER(2))  -- illegal
```

Examples

Some examples of cursor declarations follow:

```
CURSOR c1 IS SELECT empno, ename, job, sal FROM emp
   WHERE sal > 2000;
CURSOR c2 RETURN dept%ROWTYPE IS
   SELECT * FROM dept WHERE deptno = 10;
CURSOR c3 (start_date DATE) IS
   SELECT empno, sal FROM emp WHERE hiredate > start_date;
```

Related Topics

CLOSE Statement, FETCH Statement, OPEN Statement, SELECT INTO Statement
DELETE Statement

The DELETE statement removes entire rows of data from a specified table or view. For a full description of the DELETE statement, see Oracle9i SQL Reference.

Syntax

delete_statement

DELETE

FROM

table_reference

WHERE

search_condition

CURRENT OF cursor_name

returning_clause

RETURNING single_row_expression INTO variable_name

RETURNING multiple_row_expression INTO BULK COLLECT INTO collection_name

schema_name

db_table_name

view_name

@dblink_name
Keyword and Parameter Description

**alias**
This is another (usually short) name for the referenced table or view and is typically used in the **WHERE** clause.

**BULK COLLECT**
This clause 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 **RETURNING INTO** list. The corresponding columns must store scalar (not composite) values. For more information, see “Reducing Loop Overhead for Collections with Bulk Binds” on page 5-32.

**returning_clause**
This clause lets you return values from the deleted rows, thereby eliminating the need to **SELECT** the rows beforehand. You can retrieve the column values into variables and/or host variables, or into collections and/or host arrays. However, you cannot use the **RETURNING** clause for remote or parallel deletes.

**subquery**
This is a **SELECT** statement that provides a set of rows for processing. Its syntax is like that of **select_into_statement** without the **INTO** clause. See “**SELECT INTO** Statement” on page 13-164.

**table_reference**
This specifies a table or view, which must be accessible when you execute the **DELETE** statement, and for which you must have **DELETE** privileges.

**TABLE (subquery2)**
The operand of **TABLE** is a **SELECT** statement that returns a single column value, which must be a nested table. Operator **TABLE** informs Oracle that the value is a collection, not a scalar value.
**WHERE CURRENT OF cursor_name**

This clause refers to the latest row processed by the FETCH statement associated with the cursor identified by cursor_name. The cursor must be FOR UPDATE and must be open and positioned on a row. If the cursor is not open, the CURRENT OF clause causes an error.

If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception NO_DATA_FOUND.

**WHERE search_condition**

This clause conditionally chooses rows to be deleted from the referenced table or view. Only rows that meet the search condition are deleted. If you omit the WHERE clause, all rows in the table or view are deleted.

**Usage Notes**

You can use the DELETE WHERE CURRENT OF statement after a fetch from an open cursor (this includes implicit fetches executed in a cursor FOR loop), provided the associated query is FOR UPDATE. This statement deletes the current row; that is, the one just fetched.

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND, and %ROWCOUNT let you access useful information about the execution of a DELETE statement.

**Examples**

The following statement deletes from the bonus table all employees whose sales were below quota:

```
DELETE FROM bonus WHERE sales_amt < quota;
```

The following statement returns two column values from a deleted row into local variables:

```
DECLARE
  my_empno emp.empno%TYPE;
  my_ename emp.ename%TYPE;
  my_job   emp.job%TYPE;
BEGIN
  ...
  DELETE FROM emp WHERE empno = my_empno
    RETURNING ename, job INTO my_ename, my_job;
END;
```
You can combine the BULK COLLECT clause with a FORALL statement, in which case, the SQL engine bulk-binds column values incrementally. In the following example, if collection `depts` has 3 elements, each of which causes 5 rows to be deleted, then collection `enums` has 15 elements when the statement completes:

```sql
FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp WHERE deptno = depts(j)
    RETURNING empno BULK COLLECT INTO enums;
```

The column values returned by each execution are added to the values returned previously.

**Related Topics**

FETCH Statement, SELECT Statement
The pragma EXCEPTION_INIT associates an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it instead of using the OTHERS handler. For more information, see "Associating a PL/SQL Exception with a Number: Pragma EXCEPTION_INIT" on page 7-8.

**Syntax**

```
exception_init_pragma
```

<table>
<thead>
<tr>
<th>Keyword and Parameter Description</th>
</tr>
</thead>
</table>

**error_number**

This is any valid Oracle error number. These are the same error numbers returned by the function SQLCODE.

**exception_name**

This identifies a user-defined exception previously declared within the current scope.

**PRAGMA**

This keyword signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler.

**Usage Notes**

You can use EXCEPTION_INIT in the declarative part of any PL/SQL block, subprogram, or package. The pragma must appear in the same declarative part as its associated exception, somewhere after the exception declaration.

Be sure to assign only one exception name to an error number.
Example

The following pragma associates the exception `deadlock_detected` with Oracle error 60:

```
DECLARE
  deadlock_detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
  ...
  EXCEPTION
    WHEN deadlock_detected THEN
      -- handle the error
  ...
END;
```

Related Topics

AUTONOMOUS_TRANSACTION Pragma, Exceptions, SQLCODE Function
Exceptions

An exception is a runtime error or warning condition, which can be predefined or user-defined. Predefined exceptions are raised implicitly (automatically) by the runtime system. User-defined exceptions must be raised explicitly by `RAISE` statements. To handle raised exceptions, you write separate routines called exception handlers. For more information, see Chapter 7.

Syntax

```
exception_name  EXCEPTION ;
```

```
exception_handler

WHEN \exception_name\ OR \exception_name\ OTHERS THEN statement
```

Keyword and Parameter Description

**exception_name**

This identifies a predefined exception such as `ZERO_DIVIDE`, or a user-defined exception previously declared within the current scope.

**OTHERS**

This keyword stands for all the exceptions not explicitly named in the exception-handling part of the block. The use of `OTHERS` is optional and is allowed only as the last exception handler. You cannot include `OTHERS` in a list of exceptions following the keyword `WHEN`.

**statement**

This is an executable statement. For the syntax of `statement`, see "Blocks" on page 13-11.
WHEN
This keyword 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 "Predefined PL/SQL Exceptions" on page 7-4. 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.

An exception should be raised 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

The following PL/SQL block has two exception handlers:

```pl/sql
DECLARE
  bad_emp_id  EXCEPTION;
  bad_acct_no EXCEPTION;
  ...
BEGIN
  ...
EXCEPTION
  WHEN bad_emp_id OR bad_acct_no THEN  -- user-defined
    ROLLBACK;
  WHEN ZERO_DIVIDE THEN  -- predefined
    INSERT INTO inventory VALUES (part_number, quantity);
    COMMIT;
END;
```

Related Topics

Blocks, EXCEPTION_INIT Pragma, RAISE Statement
The **EXECUTE IMMEDIATE** statement prepares (parses) and immediately executes a dynamic SQL statement or anonymous PL/SQL block. For more information, see Chapter 11.

**Syntax**

```
execute_immediate_statement
```

**Keyword and Parameter Description**

**bind_argument**

This can be an expression whose value is passed to the dynamic SQL statement or PL/SQL block, or it can be a variable that stores a value returned by the dynamic SQL statement or PL/SQL block.
**define_variable_name**
This identifies a variable that stores a selected column value.

**dynamic_string**
This is a string literal, variable, or expression that represents a SQL statement or PL/SQL block.

**INTO ...**
Used only for single-row queries, this clause specifies the variables or record into which column values are retrieved. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.

**record_name**
This identifies a user-defined or %ROWTYPE record that stores a selected row.

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

**USING ...**
This clause specifies a list of input and/or output bind arguments. If you do not specify a parameter mode, it defaults to IN.

**Usage Notes**
Except for multi-row queries, the dynamic string can contain any SQL statement (without the terminator) or any PL/SQL block (with the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement.

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

At run time, bind arguments replace corresponding placeholders in the dynamic string. So, every placeholder must be associated with a bind argument in the USING clause and/or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL). To pass nulls to the dynamic string, you must use a workaround. See "Passing Nulls" on page 11-17.

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

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

Examples

The following PL/SQL block contains several examples of dynamic SQL:

```plsql
DECLARE
    sql_stmt    VARCHAR2(200);
    plsql_block VARCHAR2(500);
    emp_id      NUMBER(4) := 7566;
    salary      NUMBER(7,2);
    dept_id     NUMBER(2) := 50;
    dept_name   VARCHAR2(14) := 'PERSONNEL';
    location    VARCHAR2(13) := 'DALLAS';
    emp_rec     emp%ROWTYPE;
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

    sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
    EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;

    sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
    EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;

    plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
    EXECUTE IMMEDIATE plsql_block USING 7788, 500;

    sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
        RETURNING sal INTO :2';
    EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;
```
EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
    USING dept_id;

EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
END;

Related Topics

OPEN-FOR-USING Statement
EXIT Statement

You use the EXIT statement to exit a loop. The EXIT statement has two forms: the unconditional EXIT and the conditional EXIT WHEN. With either form, you can name the loop to be exited. For more information, see "Iterative Control: LOOP and EXIT Statements" on page 4-9.

Syntax

```
EXIT label_name [WHEN boolean_expression]
```

Keyword and Parameter Description

**boolean_expression**

This is an expression that yields the Boolean value TRUE, FALSE, or NULL. It is evaluated with each iteration of the loop in which the EXIT WHEN statement appears. If the expression yields TRUE, the current loop (or the loop labeled by label_name) is exited immediately. For the syntax of boolean_expression, see "Expressions" on page 13-72.

**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**

This identifies the loop to be exited. You can exit not only the current loop but any enclosing labeled loop.

Usage Notes

The EXIT statement can be used only inside a loop. PL/SQL lets you code an infinite loop. For example, the following loop will never terminate normally:

```plsql
WHILE TRUE LOOP ... END LOOP;
```

In such cases, you must use an EXIT statement to exit the 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

The EXIT statement in the following example is illegal because you cannot exit from a block directly; you can exit only from a loop:

```plsql
DECLARE
    amount  NUMBER;
    maximum NUMBER;
BEGIN
    ...
    BEGIN
        ...
        IF amount >= maximum THEN
            EXIT;  -- illegal; use RETURN instead
        END IF;
    END;
END;
```

The following loop normally executes ten times, but it will exit prematurely if there are less than ten rows to fetch:

```plsql
FOR i IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    total_comm := total_comm + emp_rec.comm;
END LOOP;
```

The following example illustrates the use of loop labels:

```plsql
<<outer>>
FOR i IN 1..10 LOOP
    ...
<<inner>>
    FOR j IN 1..100 LOOP
        ...
        EXIT outer WHEN ... -- exits both loops
    END LOOP inner;
END LOOP outer;
```

Related Topics

Expressions, LOOP Statements
Expressions

An expression is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression is a single variable.

The PL/SQL compiler determines the datatype of an expression from the types of the variables, constants, literals, and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results. For more information, see "Expressions and Comparisons" on page 2-23.

Syntax

expression

boolean_expression

character_expression
date_expression
numeric_expression

boolean_expression

boolean_constant_name
boolean_function_call
boolean_literal
boolean_variable_name
other_boolean_form

boolean_expression

AND
OR
NOT

boolean_constant_name
boolean_function_call
boolean_literal
boolean_variable_name
other_boolean_form
Expressions

other boolean form

character expression

PL/SQL Language Elements 13-73
Expressions

cursor_name
cursor_variable_name
host_cursor_variable_name
SQL
% ROWCOUNT

SQL
% BULK_ROWCOUNT
index
indicator_name

host_variable_name
numeric_constant_name
numeric_function_call
numeric_literal
numeric_variable_name

numeric_subexpression

collection_name

COUNT
FIRST
LAST
LIMIT
NEXT
PRIOR

** exponent
Keyword and Parameter Description

**BETWEEN**
This comparison operator tests whether a value lies in a specified range. It means "greater than or equal to low value and less than or equal to high value.”

**boolean_constant_name**
This identifies a constant of type BOOLEAN, which must be initialized to the value TRUE, FALSE, or NULL. Arithmetic operations on Boolean constants are illegal.

**boolean_expression**
This is an expression that yields the Boolean value TRUE, FALSE, or NULL.
**boolean_function_call**
This is any function call that returns a Boolean value.

**boolean_literal**
This is the predefined value 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**
This identifies 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 illegal.

**%BULK_ROW_COUNT**
Designed for use with the FORALL statement, this is a composite attribute of the implicit cursor SQL. For more information, see "SQL Cursor" on page 13-173.

**character_constant_name**
This identifies 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**
This is an expression that yields a character or character string.

**character_function_call**
This is a function call that returns a character value or a value implicitly convertible to a character value.

**character_literal**
This is a literal that represents a character value or a value implicitly convertible to a character value.

**character_variable_name**
This identifies a previously declared variable that stores a character value.
**collection_name**
This identifies a collection (nested table, index-by table, or varray) previously declared within the current scope.

**cursor_name**
This identifies an explicit cursor previously declared within the current scope.

**cursor_variable_name**
This identifies a PL/SQL cursor variable previously declared within the current scope.

**date_constant_name**
This identifies 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**
This is an expression that yields a date/time value.

**date_function_call**
This is a function call that returns a date value or a value implicitly convertible to a date value.

**date_literal**
This is a literal that represents a date value or a value implicitly convertible to a date value.

**date_variable_name**
This identifies a previously declared variable that stores a date value.

**EXISTS, COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR**
These are collection methods. When appended to the name of a collection, these methods return useful information. For example, EXIST(n) returns TRUE if the nth element of a collection exists. Otherwise, EXIST(n) returns FALSE. For more information, see "Collection Methods" on page 13-23.

**exponent**
This is an expression that must yield a numeric value.
%FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT
These are cursor attributes. When appended to the name of a cursor or cursor variable, these attributes return useful information about the execution of a multi-row query. You can also append them to the implicit cursor SQL.

host_cursor_variable_name
This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Host cursor variables must be prefixed with a colon.

host_variable_name
This identifies a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host variable must be implicitly convertible to the appropriate PL/SQL datatype. Also, host variables must be prefixed with a colon.

IN
This comparison operator 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

yield FALSE if the set contains a null.

index
This is a numeric expression that must yield a value of type BINARY_INTEGER or a value implicitly convertible to that datatype.

indicator_name
This identifies 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
This comparison operator returns the Boolean value TRUE if its operand is null, or FALSE if its operand is not null.
LIKE
This comparison operator 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
These are logical operators, which follow the tri-state logic of Table 2–2 on page 2-25. 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 2-25.

NULL
This keyword 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
This identifies 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
This is an expression that yields an integer or real value.

numeric_function_call
This is a function call that returns a numeric value or a value implicitly convertible to a numeric value.

numeric_literal
This is a literal that represents a number or a value implicitly convertible to a number.

numeric_variable_name
This identifies a previously declared variable that stores a numeric value.
pattern
This is 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.

relational_operator
This operator lets you compare expressions. For the meaning of each operator, see "Comparison Operators" on page 2-26.

SQL
This identifies a cursor opened implicitly by Oracle to process a SQL data manipulation statement. The implicit cursor SQL always refers to the most recently executed SQL statement.

+, -, /, *, **
These symbols are the addition, subtraction, division, multiplication, and exponentiation operators, respectively.

||
This is the concatenation operator. As the following example shows, the result of concatenating string1 with string2 is a character string that contains string1 followed by string2:

'Good' || ' morning!' = 'Good morning!'

The next example shows that nulls have no effect on the result of a concatenation:

'suit' || NULL || 'case' = 'suitcase'

A null string (' '), which is zero characters in length, is treated like a null.

Usage Notes
In a Boolean expression, you can only compare values that have compatible datatypes. For more information, see "Datatype Conversion" on page 3-24.

In conditional control statements, if a Boolean expression yields TRUE, its associated sequence of statements is executed. But, if the expression yields 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 yield a null. The
value of a Boolean expression can be assigned only to Boolean variables, not to host variables or database columns. Also, datatype conversion to or from type BOOLEAN is not supported.

You can use the addition and subtraction operators to increment or decrement a date value, as the following examples show:

```plsql
hire_date := '10-MAY-95';
hire_date := hire_date + 1;  -- makes hire_date '11-MAY-95'
hire_date := hire_date - 5;  -- makes hire_date '06-MAY-95'
```

When PL/SQL evaluates a boolean expression, NOT has the highest precedence, AND has the next-highest precedence, and OR has the lowest precedence. However, you can use parentheses to override the default operator precedence.

Within an expression, operations occur in their predefined order of precedence. From first to last (top to bottom), the default order of operations is

- parentheses
- exponents
- unary operators
- multiplication and division
- 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**

Several examples of expressions follow:

```plsql
(a + b) > c               -- Boolean expression
NOT finished             -- Boolean expression
TO_CHAR(acct_no)          -- character expression
'Fat ' || 'cats'          -- character expression
'15-NOV-95'              -- date expression
MONTHS_BETWEEN(d1, d2)   -- date expression
pi * r**2                 -- numeric expression
emp_cv%ROWCOUNT           -- numeric expression
```
Expressions

Related Topics
Assignment Statement, Constants and Variables, EXIT Statement, IF Statement, LOOP Statements
FETCH Statement

The FETCH statement retrieves rows of data one at a time from the result set of a multi-row query. The data is stored in variables or fields that correspond to the columns selected by the query. For more information, see "Managing Cursors" on page 6-6.

Syntax

```plaintext
fetch_statement

FETCH
  cursor_name
  cursor_variable_name
  host_cursor_variable_name

INTO
  variable_name
  record_name

BULK COLLECT INTO
  collection_name
  host_array_name

LIMIT numeric_expression
```

Keyword and Parameter Description

**BULK COLLECT**
This clause 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**
This identifies 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.
**FETCH Statement**

- **cursor_name**
  This identifies an explicit cursor previously declared within the current scope.

- **cursor_variable_name**
  This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope.

- **host_array_name**
  This identifies an array (declared in a PL/SQL host environment and passed to PL/SQL as a bind variable) into which column values are bulk fetched. For each query `select_item`, there must be a corresponding, type-compatible array in the list. Host arrays must be prefixed with a colon.

- **host_cursor_variable_name**
  This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

- **LIMIT**
  This optional clause, allowed only in bulk (not scalar) `FETCH` statements, lets you limit the number of rows fetched from the database.

- **record_name**
  This identifies 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**
  This identifies a previously declared 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 multi-row query.
Any variables in the \texttt{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 \texttt{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 \texttt{FETCH} past the last row in the result set, the values of the target fields or variables are indeterminate and the \%NOTFOUND attribute yields \texttt{TRUE}.

PL/SQL makes sure the return type of a cursor variable is compatible with the \texttt{INTO} clause of the \texttt{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 \texttt{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 \texttt{IN} or \texttt{IN OUT} mode. However, if the subprogram also opens the cursor variable, you must specify the \texttt{IN OUT} mode.

Eventually, the \texttt{FETCH} statement must fail to return a row, so when that happens, no exception is raised. To detect the failure, you must use the cursor attribute \%FOUND or \%NOTFOUND.

PL/SQL raises the predefined exception \texttt{INVALID_CURSOR} if you try to fetch from a closed or never-opened cursor or cursor variable.

\textbf{Examples}

The following example shows that any variables in the query associated with a cursor are evaluated only when the cursor is opened:

\begin{verbatim}
DECLARE
  my_sal  NUMBER(7,2);
  n      INTEGER(2) := 2;
  CURSOR emp_cur IS SELECT n*sal FROM emp;
\end{verbatim}
FETCH Statement

BEGIN
  OPEN emp_cur;  -- n equals 2 here
  LOOP
    FETCH emp_cur INTO my_sal;
    EXIT WHEN emp_cur%NOTFOUND;
    -- process the data
    n := n + 1;  -- does not affect next FETCH; sal will be
    multiplied by 2
  END LOOP;
END;

In the following example, you fetch rows one at a time from the cursor variable
emp_cv into the user-defined record emp_rec:

DECLARE
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  emp_cv  EmpCurTyp;
  emp_rec emp%ROWTYPE;
BEGIN
  LOOP
    FETCH emp_cv INTO emp_rec;
    EXIT WHEN emp_cv%NOTFOUND;
    ...
  END LOOP;
END;

The BULK COLLECT clause lets you bulk-bind entire columns of Oracle data. That
way, you can fetch all rows from the result set at once. In the following example,
you bulk-fetch from a cursor into a collection:

DECLARE
  TYPE NameList IS TABLE OF emp.ename%TYPE;
  names NameList;
  CURSOR c1 IS SELECT ename FROM emp WHERE job = 'CLERK';
BEGIN
  OPEN c1;
  FETCH c1 BULK_collect INTO names;
  ...
  CLOSE c1;
END;
In the example below, you use the LIMIT clause. With each iteration of the loop, the FETCH statement fetches 100 rows (or less) into index-by table acct_ids. The previous values are overwritten.

DECLARE
   TYPE NumList IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
   CURSOR c1 IS SELECT acct_id FROM accounts;
   acct_ids NumList;
   rows NATURAL := 100;  -- set limit
BEGIN
   OPEN c1;
   LOOP
      /* The following statement fetches 100 rows (or less). */
      FETCH c1 BULK COLLECT INTO acct_ids LIMIT rows;
      EXIT WHEN c1%NOTFOUND;
      ...  
   END LOOP;
   CLOSE c1;
END;

Related Topics

CLOSE Statement, Cursors, Cursor Variables, LOOP Statements, OPEN Statement, OPEN-FOR Statement
FORALL Statement

The FORALL statement instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is not a FOR loop. For more information, see "Reducing Loop Overhead for Collections with Bulk Binds" on page 5-32.

Syntax

```
forall_statement

FORALL index_name IN lower_bound .. upper_bound

sql_statement
```

Keyword and Parameter Description

**index_name**

This is 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. So, another variable with the same name cannot be referenced inside the statement. Inside a FORALL statement, index_name cannot appear in expressions and cannot be assigned a value.

**lower_bound .. upper_bound**

These are expressions that must yield number, which, if necessary, PL/SQL rounds to the nearest integer. The integers must specify a valid range of consecutive index numbers. The SQL engine executes the SQL statement once for each index number in the range. The expressions are evaluated only when the FORALL statement is first entered.

**SAVE EXCEPTIONS**

These optional keywords cause the FORALL loop to continue even if some DML operations fail. 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.
**sql_statement**

This must be an INSERT, UPDATE, or DELETE statement that references collection elements.

**Usage Notes**

The SQL statement can reference more than one collection. However, the PL/SQL engine bulk-binds only subscripted collections.

All collection elements in the specified range must exist. If an element is missing or was deleted, you get an error.

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 executions are not rolled back.

**Example**

The following example shows that you can use the lower and upper bounds to bulk-bind arbitrary slices of a collection:

```sql
DECLARE
    TYPE NumList IS VARRAY(15) OF NUMBER;
    depts NumList := NumList();
BEGIN
    -- fill varray here
    ...
    FOR ALL j IN 6..10  -- bulk-bind middle third of varray
        UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
END;
```

Remember, the PL/SQL engine bulk-binds only subscripted collections. So, in the following example, it does not bulk-bind the collection `sals`, which is passed to the function `median`:

```sql
FOR ALL i IN 1..20
    INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);
```

**Related Topics**

BULK COLLECT Clause
A function is a subprogram that can take parameters and be invoked. Generally, you use a function to compute a value. A function has two parts: the specification and the body. The specification (spec for short) begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the datatype of the return value. Parameter declarations are optional. Functions that take no parameters are written without parentheses. The function body begins with the keyword IS (or AS) and ends with the keyword END followed by an optional function name.

The function body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. These items are local and cease to exist when you exit the function. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains handlers that deal with exceptions raised during execution. For more information, see "Understanding PL/SQL Functions" on page 8-6.
Syntax

```
function_spec
  FUNCTION function_name (parameter_declaration) RETURN datatype
  CREATE OR REPLACE
  AUTHID CURRENT_USER DEFINER PARALLEL_ENABLE DETERMINISTIC
  IS
  AS
```
Keyword and Parameter Description

**AUTHID**
This clause determines whether a stored function executes with the privileges of its owner (the default) or current user and whether its unqualified references to schema objects are resolved in the schema of the owner or current user. You can override the default behavior by specifying `CURRENT_USER`. For more information, see "Invoker Rights Versus Definer Rights" on page 8-47.

**CREATE**
The optional `CREATE` clause lets you create standalone functions, which are stored in the Oracle database. You can execute the `CREATE` statement interactively from SQL*Plus or from a program using native dynamic SQL.
datatype
This is a type specifier. For the syntax of datatype, see "Constants and Variables" on page 13-38.

DETERMINISTIC
This hint helps the optimizer avoid redundant function calls. If a stored function was called previously with the same arguments, the optimizer can elect to use the previous result. The function result should not depend on the state of session variables or schema objects. Otherwise, results might vary across calls. Only DETERMINISTIC functions can be called from a function-based index or a materialized view that has query-rewrite enabled. For more information, see the statements CREATE INDEX and CREATE MATERIALIZED VIEW in Oracle9i SQL Reference.

exception_handler
This associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of exception_handler, see "Exceptions" on page 13-63.

expression
This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of expression is assigned to the parameter. The value and the parameter must have compatible datatypes.

function_name
This identifies a user-defined function.

IN, OUT, IN OUT
These parameter modes define the behavior of formal parameters. An IN parameter lets you pass values to the subprogram being called. An OUT parameter lets you return values to the caller of the subprogram. An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

item_declaration
This declares a program object. For the syntax of item_declaration, see "Blocks" on page 13-11.
**NOCOPY**
This is a compiler hint (not directive), which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference instead of by value (the default). For more information, see "Passing Large Data Structures with the NOCOPY Compiler Hint" on page 8-17.

**PARALLEL_ENABLE**
This option declares that a stored function can be used safely in the slave sessions of parallel DML evaluations. The state of a main (logon) session is never shared with slave sessions. Each slave session has its own state, which is initialized when the session begins. The function result should not depend on the state of session (static) variables. Otherwise, results might vary across sessions.

**parameter_name**
This identifies a formal parameter, which is a variable declared in a function spec and referenced in the function body.

**PRAGMA AUTONOMOUS_TRANSACTION**
This pragma instructs the PL/SQL compiler to mark a function as autonomous (independent). An autonomous transaction is an independent transaction started by another transaction, the main transaction. Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction. For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-53.

**procedure_declaration**
This declares a procedure. For the syntax of procedure_declaration, see "Procedures" on page 13-143.

**RETURN**
This keyword introduces the RETURN clause, which specifies the datatype of the return value.

**type_definition**
This specifies a user-defined datatype. For the syntax of type_definition, see "Blocks" on page 13-11.
This operator or keyword lets you initialize IN parameters to default values.

Usage Notes

As the following example shows, a function is called as part of an expression:

```plaintext
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

To be callable from SQL statements, a stored function must obey certain rules meant to control side effects. See "Controlling Side Effects of PL/SQL Subprograms" on page 8-9.

In a function, there must be at least one execution path that leads to a RETURN statement. Otherwise, you get a function returned without value error at run time. Also, the RETURN statement must contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable.

You can write the function spec and body as a unit. Or, you can separate the function spec from its body. That way, you can hide implementation details by placing the function in a package. You can define functions in a package body without declaring their specs in the package spec. However, such functions can be called only from inside the package.

Inside a function, an IN parameter acts like a constant. So, you cannot assign it a value. An OUT parameter acts like a local variable. So, you can change its value and reference the value in any way. An IN OUT parameter acts like an initialized variable. So, you can assign it a value, which can be assigned to another variable.

For summary information about the parameter modes, see Table 8–1 on page 8-16.

Avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take zero or more parameters and return a single value. Also, functions should be free from side effects, which change the values of variables not local to the subprogram.

Example

The following function returns the balance of a specified bank account:

```plaintext
FUNCTION balance (acct_id INTEGER) RETURN REAL IS
  acct_bal  REAL;
BEGIN
  SELECT bal INTO acct_bal FROM accts WHERE acctno = acct_id;
  RETURN acct_bal;
```
END balance;

Related Topics

Collection Methods, Packages, Procedures
GOTO Statement

The GOTO statement branches unconditionally to a statement label or block label. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. The GOTO statement transfers control to the labelled statement or block. For more information, see “GOTO Statement” on page 4-17.

Syntax

```sql
labelDeclaration
    GOTO labelName;
```

Keyword and Parameter Description

**label_name**

This is an undeclared identifier that labels an executable statement or a PL/SQL block. You use a GOTO statement to transfer control to the statement or block following `<label_name>`.

Usage Notes

Some possible destinations of a GOTO statement are illegal. In particular, a GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block. For example, the following GOTO statement is illegal:

```sql
BEGIN
    ...
    GOTO update_row; -- illegal branch into IF statement
    ...
    IF valid THEN
        ...
        update_row>
        UPDATE emp SET ...
    END IF;
```
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

A GOTO label cannot precede just any keyword. It must precede an executable statement or a PL/SQL block. For example, the following GOTO statement is illegal:

```sql
FOR ctr IN 1..50 LOOP
    DELETE FROM emp WHERE ...
    IF SQL%FOUND THEN
        GOTO end_loop;  -- illegal
        END IF;
    ...
<<end_loop>>
END LOOP;   -- not an executable statement
```

To debug the last example, simply add the NULL statement, as follows:

```sql
FOR ctr IN 1..50 LOOP
    DELETE FROM emp WHERE ...
    IF SQL%FOUND THEN
        GOTO end_loop;
        END IF;
    ...
<<end_loop>>
NULL;  -- an executable statement that specifies inaction
END LOOP;
```
The IF statement lets you execute a sequence of statements conditionally. Whether the sequence is executed or not depends on the value of a Boolean expression. For more information, see “Conditional Control: IF and CASE Statements” on page 4-2.

**Syntax**

```
if_statement

IF boolean_expression THEN statement

ELSIF boolean_expression THEN statement

ELSE statement

END IF
```

**Keyword and Parameter Description**

**boolean_expression**

This is an expression that yields the Boolean value TRUE, FALSE, or NULL. It is associated with a sequence of statements, which is executed only if the expression yields TRUE.

**ELSE**

If control reaches this keyword, the sequence of statements that follows it is executed.

**ELSIF**

This keyword introduces a Boolean expression to be evaluated if the expression following IF and all the expressions following any preceding ELSIFs yield FALSE or NULL.
**THEN**

This keyword associates the Boolean expression that precedes it with the sequence of statements that follows it. If the expression yields `TRUE`, the associated sequence of statements is 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 yields `TRUE`. If the expression yields `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 yields `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 yields `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 yields `TRUE`, its associated sequence of statements is executed and control passes to the next statement. If all expressions yield `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

In the example below, if `shoe_count` has a value of 10, both the first and second Boolean expressions yield `TRUE`. Nevertheless, `order_quantity` is assigned the proper value of 50 because processing of an `IF` statement stops after an expression yields `TRUE` and its associated sequence of statements is executed. The expression associated with `ELSIF` is never evaluated and control passes to the `INSERT` statement.

```sql
IF shoe_count < 20 THEN
    order_quantity := 50;
ELSIF shoe_count < 30 THEN
    order_quantity := 20;
ELSE
    order_quantity := 10;
END IF;

INSERT INTO purchase_order VALUES (shoe_type, order_quantity);
```

In the following example, depending on the value of `score`, one of two status messages is inserted into the `grades` table:

```sql
IF score < 70 THEN
    fail := fail + 1;
    INSERT INTO grades VALUES (student_id, 'Failed');
ELSE
    pass := pass + 1;
    INSERT INTO grades VALUES (student_id, 'Passed');
END IF;
```

Related Topics

Expressions
The `INSERT` statement adds new rows of data to a specified database table. For a full description of the `INSERT` statement, see *Oracle9i SQL Reference*.

**Syntax**

```
insert_statement

INSERT INTO
    table_reference
    subquery
    TABLE
    subquery2
    alias

column_name

VALUES
    sql_expression
    returning_clause
    subquery3
```

**Keyword and Parameter Description**

- **alias**
  This is another (usually short) name for the referenced table or view.

- **column_name[, column_name]...**
  This identifies a list of columns in a database table or view. Column names need not appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement. However, no column name can appear more than once in the list. If the list does not include all the columns in a table, the missing columns are set to NULL or to a default value specified in the CREATE TABLE statement.
**returning_clause**
This clause lets you return values from inserted rows, thereby eliminating the need to `SELECT` the rows afterward. You can retrieve the column values into variables and/or host variables, or into collections and/or host arrays. However, you cannot use the `RETURNING` clause for remote or parallel inserts. For the syntax of `returning_clause`, see "DELETE Statement" on page 13-57.

**sql_expression**
This is any expression valid in SQL. For more information, see Oracle9i SQL Reference.

**subquery**
This is a `SELECT` statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 13-164.

**subquery3**
This is a `SELECT` statement that returns a value or set of values. As many rows are added to the table as are returned by the subquery. It must return a value for every column in the column list or for every column in the table if there is no column list.

**table_reference**
This identifies a table or view that must be accessible when you execute the `INSERT` statement, and for which you must have `INSERT` privileges. For the syntax of `table_reference`, see "DELETE Statement" on page 13-57.

**TABLE (subquery2)**
The operand of `TABLE` is a `SELECT` statement that returns a single column value, which must be a nested table. Operator `TABLE` informs Oracle that the value is a collection, not a scalar value.

**VALUES (…)**
This clause assigns the values of expressions to corresponding columns in the column list. If there is no column list, the first value is inserted into the first column defined by the `CREATE TABLE` statement, the second value is inserted into the second column, and so on. There must be only one value for each column in the column list. Also, the datatypes of the values being inserted must be compatible with the datatypes of corresponding columns in the column list.
Usage Notes

Character and date literals in the VALUES list must be enclosed by single quotes ('). Numeric literals are not enclosed by quotes.

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of an INSERT statement.

Examples

The following examples show various forms of INSERT statement:

```sql
INSERT INTO bonus SELECT ename, job, sal, comm FROM emp
    WHERE comm > sal * 0.25;
...
INSERT INTO emp (empno, ename, job, sal, comm, deptno)
    VALUES (4160, 'STURDEVIN', 'SECURITY GUARD', 2045, NULL, 30);
...
INSERT INTO dept
    VALUES (my_deptno, UPPER(my_dname), 'CHICAGO');
```

Related Topics

SELECT Statement
Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. The numeric literal 135 and the string literal ‘hello world’ are examples. For more information, see "Literals" on page 2-7.
Keyword and Parameter Description

character
This is a member of the PL/SQL character set. For more information, see "Character Set" on page 2-2.

digit
This is one of the numerals 0 .. 9.

TRUE, FALSE, NULL
This is a predefined Boolean value.

Usage Notes
Two kinds of numeric literals can be used in arithmetic expressions: integers and reals. Numeric literals must be separated by punctuation. Spaces can be used in addition to the punctuation.

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. So, for example, PL/SQL considers the literals ‘Q’ and ‘q’ to be different.

A string literal is a sequence of zero or more characters enclosed by single quotes. The null string (‘ ’) contains zero characters. To represent an apostrophe within a string, write two single quotes. PL/SQL is case sensitive within string literals. So, for example, PL/SQL considers the literals ‘white’ and ‘White’ to be different.

Also, trailing blanks are significant within string literals, so ‘abc’ and ‘abc ’ are different. Trailing blanks in a literal are never trimmed.

The Boolean values TRUE and FALSE cannot be inserted into a database column.
Examples

Several examples of numeric literals follow:

25   6.34   7E2   25e-03   .1   1.   +17   -4.4

Several examples of character literals follow:

'H'   'a'   ' '   '9'   ']'   'g'

A few examples of string literals follow:

'$5,000'
'
02-AUG-87'
'Don’t leave without saving your work.’

Related Topics

Constants and Variables, Expressions
LOCK TABLE Statement

The `LOCK TABLE` statement lets you lock entire database tables in a specified lock mode. That enables you to share or deny access to tables while maintaining their integrity. For more information, see "Using LOCK TABLE" on page 6-51.

Syntax

```
lock_table_statement
```

```
LOCK TABLE table_reference IN lock_mode MODE NOWAIT ;
```

Keyword and Parameter Description

**table_reference**

This identifies a table or view that must be accessible when you execute the `LOCK TABLE` statement. For the syntax of `table_reference`, see "DELETE Statement" on page 13-57.

**lock_mode**

This parameter specifies the lock mode. It must be one of the following: ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE, SHARE, SHARE ROW EXCLUSIVE, or EXCLUSIVE.

**NOWAIT**

This optional keyword tells Oracle not to wait if the table has been locked by another user. Control is immediately returned to your program, so it can do other work before trying again to acquire the lock.

Usage Notes

If you omit the keyword NOWAIT, Oracle waits until the table is available; the wait has no set limit. Table locks are released when your transaction issues a commit or rollback.

A table lock never keeps other users from querying a table, and a query never acquires a table lock.
If your program includes SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as SELECT, INSERT, UPDATE, or DELETE.

Example

The following statement locks the accts table in shared mode:

LOCK TABLE accts IN SHARE MODE;

Related Topics

COMMIT Statement, ROLLBACK Statement
LOOP statements execute a sequence of statements multiple times. The loop encloses the sequence of statements that is to be repeated. PL/SQL provides four kinds of loop statements: basic loop, WHILE loop, FOR loop, and cursor FOR loop. For more information, see “Iterative Control: LOOP and EXIT Statements” on page 4-9.

Syntax

basic_loop_statement

while_loop_statement

for_loop_statement
Keyword and Parameter Description

**basic_loop_statement**
The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords `LOOP` and `END LOOP`. With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use the `EXIT`, `GOTO`, or `RAISE` statement to complete the loop. A raised exception will also complete the loop.

**boolean_expression**
This is an expression that yields the Boolean value `TRUE`, `FALSE`, or `NULL`. It is associated with a sequence of statements, which is executed only if the expression yields `TRUE`. For the syntax of `boolean_expression`, see "Expressions" on page 13-72.

**cursor_for_loop_statement**
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 have been processed.
**cursor_name**
This identifies 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**
This identifies a cursor parameter; that is, a variable declared as the formal parameter of a cursor. (For the syntax of `cursor_parameter_declaration`, see "Cursors" 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**
Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. Numeric FOR 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 sequence of statements in the loop 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**
This is an undeclared identifier that names the loop index (sometimes called a loop counter). Its scope is the loop itself. Therefore, you cannot reference the index outside the loop.

The implicit declaration of `index_name` overrides any other declaration outside the loop. So, another variable with the same name cannot be referenced inside the loop unless a label is used, as follows:

```
<<main>>
DECLARE
  num NUMBER;
BEGIN
  ...
  FOR num IN 1..10 LOOP
    IF main.num > 5 THEN -- refers to the variable num, -- not to the loop index
      ...
    END IF;
  END LOOP;
END main;
```
Inside a loop, its index is treated like a constant. The index can appear in expressions, but cannot be assigned a value.

**label_name**

This is an undeclared identifier that optionally labels a loop. If used, `label_name` must be enclosed by double angle brackets and must appear at the beginning of the loop. Optionally, `label_name` (not enclosed in angle brackets) can also appear at the end of the loop.

You can use `label_name` in an EXIT statement to exit the loop labelled by `label_name`. You can exit not only the current loop, but any enclosing loop.

You cannot reference the index of a **FOR** loop from a nested **FOR** loop if both indexes have the same name unless the outer loop is labeled by `label_name` and you use dot notation, as follows:

`label_name.index_name`

In the following example, you compare two loop indexes that have the same name, one used by an enclosing loop, the other by a nested loop:

```
<<outer>>
FOR ctr IN 1..20 LOOP
  ...
<<inner>>
  FOR ctr IN 1..10 LOOP
    IF outer.ctr > ctr THEN ...
  END LOOP inner;
END LOOP outer;
```

**lower_bound .. upper_bound**

These are expressions that must yield 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, as the example below shows. However, the loop counter increment (or decrement) must be 1.

```
FOR i IN -5..10 LOOP
  ...
END LOOP;
```
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 +/- 2**31. So, if a bound evaluates to a number outside that range, you get a numeric overflow error when PL/SQL attempts the assignment.

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**

This identifies 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. So, another record with the same name cannot be referenced inside the loop unless a label is used.

Fields in the record store column values from the implicitly fetched row. The fields have the same names and datatypes as their corresponding columns. To access field values, you use dot notation, as follows:

```
record_name.field_name
```

Select-items fetched from the FOR loop cursor must have simple names or, if they are expressions, must have aliases. In the following example, wages is an alias for the select item sal+comm:

```
CURSOR c1 IS SELECT empno, sal+comm wages, job ...
```

**REVERSE**

By default, iteration proceeds upward from the lower bound to the upper bound. However, if you use the keyword REVERSE, iteration proceeds downward from the upper bound to the lower bound.

An example follows:

```
FOR i IN REVERSE 1..10 LOOP  -- i starts at 10, ends at 1  
    -- statements here execute 10 times
END LOOP;
```
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**

This is 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 13-164. 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 yields TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the expression yields 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 yields 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.
Example

The following cursor FOR loop calculates a bonus, then inserts the result into a database table:

```
DECLARE
    bonus  REAL;
    CURSOR c1 IS SELECT empno, sal, comm FROM emp;
BEGIN
    FOR c1rec IN c1 LOOP
        bonus := (c1rec.sal * 0.05) + (c1rec.comm * 0.25);
        INSERT INTO bonuses VALUES (c1rec.empno, bonus);
    END LOOP;
    COMMIT;
END;
```

Related Topics

Cursors, EXIT Statement, FETCH Statement, OPEN Statement, %ROWTYPE Attribute
MERGE Statement

The MERGE statement inserts some rows and updates others in a single operation. The decision about whether to update or insert into the target table is based upon a join condition: rows already in the target table that match the join condition are updated; otherwise a row is inserted using values from a separate subquery.

For the syntax and full details of this statement, see the *Oracle9i SQL Reference*.

Usage Notes

This statement is primarily useful in data warehousing situations where large amounts of data are commonly inserted and updated.

Examples

The following example creates a bonuses table in the sample schema oe with a default bonus of 100. It then inserts into the bonuses table all employees who made sales (based on the sales_rep_id column of the oe.orders table). Finally, the Human Resources manager decides that all employees should receive a bonus:

- Those who have not made sales get a bonus of 1% of their salary.
- Those who already made sales get an increase in their bonus equal to 1% of their salary.

The MERGE statement implements these changes in one step:

```
CREATE TABLE bonuses (employee_id NUMBER, bonus NUMBER DEFAULT 100);
INSERT INTO bonuses(employee_id)
(SELECT e.employee_id FROM employees e, orders o
WHERE e.employee_id = o.sales_rep_id
GROUP BY e.employee_id);
SELECT * FROM bonuses;
EMPLOYEE_ID BONUS
---------- -----------
153 100
154 100
155 100
156 100
158 100
159 100
160 100
161 100
```
163 100
MERGE INTO bonuses D
  USING (SELECT employee_id, salary, department_id FROM employees
          WHERE department_id = 80) S
  ON (D.employee_id = S.employee_id)
  WHEN MATCHED THEN UPDATE SET D.bonus = D.bonus + S.salary*.01
  WHEN NOT MATCHED THEN INSERT (D.employee_id, D.bonus)
       VALUES (S.employee_id, S.salary*0.1);

EMPLOYEE_ID  BONUS
------------  ----------
  153         180
  154         175
  155         170
  156         200
  158         190
  159         180
  160         175
  161         170
  163         195
  157         950
  145         1400
  170         960
  179         620
  152         900
  169         1000
...
NULL Statement

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. In a construct allowing alternative actions, the NULL statement serves as a placeholder. For more information, see "NULL Statement" on page 4-21.

Syntax

```
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 has not been overlooked, but that indeed no action is necessary.

Each clause in an IF statement must contain at least one executable statement. The NULL statement meets this requirement. So, you can use the NULL statement in clauses that correspond to circumstances in which no action is taken. The NULL statement and Boolean value NULL are unrelated.

Examples

In the following example, the NULL statement emphasizes that only salespeople receive commissions:

```
IF job_title = 'SALESPERSON' THEN
    compute_commission(emp_id);
ELSE
    NULL;
END IF;
```

In the next example, the NULL statement shows that no action is taken for unnamed exceptions:

```
EXCEPTION
...
    WHEN OTHERS THEN
        NULL;
```
An object type is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called attributes. The functions and procedures that characterize the behavior of the object type are called methods.

Currently, you cannot define object types within PL/SQL. They must be created and stored in an Oracle database, where they can be shared by many programs. When you define an object type (in SQL*Plus for example) using the CREATE TYPE statement, you create an abstract template for some real-world object. The template specifies only those attributes and behaviors the object will need in the application environment.

The data structure formed by the set of attributes is public (visible to client programs). However, well-behaved programs do not manipulate it directly. Instead, they use the set of methods provided. That way, the data is kept in a proper state. At run time, when the data structure is filled with values, you have created an instance of an object type. You can create as many instances (usually called objects) as you need. For more information, see Chapter 10.

Syntax

```sql
CREATE OR REPLACE TYPE schema_name.object_type_name AS OBJECT (
  member_list
);  
attribute_name attribute_type, member_list
```
Keyword and Parameter Description

attribute_datatype
This is any Oracle datatype except LONG, LONG RAW, NCHAR, NCLOB, NVARCHAR2, ROWID, UROWID, the PL/SQL-specific types BINARY_INTEGER (and its subtypes), BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE, and %ROWTYPE, and types defined inside a PL/SQL package.

attribute_name
This identifies an object attribute. The name must be unique within the object type (but can be reused in other object types). You cannot initialize an attribute in its declaration using the assignment operator or DEFAULT clause. Also, you cannot impose the NOT NULL constraint on an attribute.
AUTHID Clause
This determines whether all member methods execute with the privileges of their
definer (the default) or invoker, and whether their unqualified references to schema
objects are resolved in the schema of the definer or invoker. For more information,
see "Invoker Rights Versus Definer Rights" on page 8-47.

call_spec
This publishes a Java method or external C function in the Oracle data dictionary. It
publishes the routine by mapping its name, parameter types, and return type to
their SQL counterparts. To learn how to write Java call specs, see Oracle9i Java Stored
Procedures Developer's Guide. To learn how to write C call specs Oracle9i Application

function_body
This defines the underlying implementation of a MEMBER function. For the syntax of
function_body, see "Functions" on page 13-90.

MAP
This keyword indicates that a method orders objects by mapping them to values of
a scalar datatype such as CHAR or REAL, which have a predefined order. PL/SQL
uses the ordering to evaluate Boolean expressions such as x > y, and to do
comparisons implied by the DISTINCT, GROUP BY, and ORDER BY clauses. A map
method returns the relative position of an object in the ordering of all such objects.
An object type can contain only one map method, which must be a parameterless
function having the return type DATE, NUMBER, VARCHAR2, or an ANSI SQL type
such as CHARACTER, INTEGER, or REAL.

MEMBER | STATIC
This keyword lets you declare a subprogram or call spec as a method in an object
type spec. The method cannot have the same name as the object type or any of its
attributes. MEMBER methods are invoked on instances, as in
instance_expression.method()

However, STATIC methods are invoked on the object type, not its instances, as in
object_type_name.method()
For each subprogram spec in an object type spec, there must be a corresponding subprogram body in the object type body. To match specs and bodies, the compiler does a token-by-token comparison of their headers. So, the headers must match word for word.

MEMBER methods accept a built-in parameter named SELF, which is an instance of the object type. Whether declared implicitly or explicitly, it is always the first parameter passed to a MEMBER method. However, STATIC methods cannot accept or reference SELF.

In the method body, SELF denotes the object whose method was invoked. For example, method transform declares SELF as an IN OUT parameter:

```
CREATE TYPE Complex AS OBJECT (  
    MEMBER FUNCTION transform (SELF IN OUT Complex) ... 
)
```

You cannot specify a different datatype for SELF. In MEMBER functions, if SELF is not declared, its parameter mode defaults to IN. However, in MEMBER procedures, if SELF is not declared, its parameter mode defaults to IN OUT. You cannot specify the OUT parameter mode for SELF.

**ORDER**

This keyword indicates that a method compares two objects. An object type can contain only one order method, which must be a function that returns a numeric result.

Every order method takes just two parameters: the built-in parameter SELF and another object of the same type. If \( c_1 \) and \( c_2 \) are Customer objects, a comparison such as \( c_1 > c_2 \) calls method match automatically. The method returns a negative number, zero, or a positive number signifying that SELF is respectively less than, equal to, or greater than the other parameter. If either parameter passed to an order method is null, the method returns a null.

**pragma_restrict_refs**

This is pragma RESTRICT_REFERENCES, which lets you check for violations of "purity" rules. To be callable from SQL statements, a member function must obey those rules, which are meant to control side effects. If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed). For the syntax of the pragma, see "RESTRICT_REFERENCES Pragma" on page 13-154 (in this context, omit the pragma terminator).
The pragma asserts that a member function does not read and/or write database tables and/or package variables. For more information about the purity rules and pragma RESTRICT_REFERENCES, see Oracle9i Application Developer’s Guide - Fundamentals.

schema_name
This qualifier identifies the schema containing the object type. If you omit schema_name, Oracle assumes the object type is in your schema.

subprogram_body
This defines the underlying implementation of a MEMBER or STATIC function or procedure. Its syntax is like that of function_body or procedure_body without the terminator. See "Functions" on page 13-90 and/or "Procedures" on page 13-143.

subprogram_spec
This declares the interface to a MEMBER or STATIC function or procedure. Its syntax is like that of function_spec or procedure_spec without the terminator. See "Functions" on page 13-90 and/or "Procedures" on page 13-143.

type_name
This identifies a user-defined object type that was defined using the datatype specifier OBJECT.

Usage Notes
Once an object type is defined and installed in the schema, you can use it to declare objects in any PL/SQL block, subprogram, or package. For example, you can use the object type to specify the datatype of an attribute, column, variable, bind variable, record field, table element, formal parameter, or function result.

Like a package, an object type has two parts: a specification and a body. The specification (spec for short) is the interface to your applications; it declares a data structure (set of attributes) along with the operations (methods) needed to manipulate the data. The body fully defines the methods, and so implements the spec.

All the information a client program needs to use the methods is in the spec. Think of the spec as an operational interface and of the body as a black box. You can debug, enhance, or replace the body without changing the spec.
Object Types

An object type encapsulates data and operations. So, you can declare attributes and methods in an object type spec, but not constants, exceptions, cursors, or types. At least one attribute is required (the maximum is 1000); methods are optional.

In an object type spec, all attributes must be declared before any methods. Only subprograms have an underlying implementation. So, if an object type spec declares only attributes and/or call specs, the object type body is unnecessary. You cannot declare attributes in the body. All declarations in the object type spec are public (visible outside the object type).

You can refer to an attribute only by name (not by its position in the object type). To access or change the value of an attribute, you use dot notation. Attribute names can be chained, which lets you access the attributes of a nested object type.

In an object type, methods can reference attributes and other methods without a qualifier. In SQL statements, calls to a parameterless method require an empty parameter list. In procedural statements, an empty parameter list is optional unless you chain calls, in which case it is required for all but the last call.

From a SQL statement, if you call a MEMBER method on a null instance (that is, SELF is null), the method is not invoked and a null is returned. From a procedural statement, if you call a MEMBER method on a null instance, PL/SQL raises the predefined exception SELF_IS_NULL before the method is invoked.

You can declare a map method or an order method but not both. If you declare either method, you can compare objects in SQL and procedural statements. However, if you declare neither method, you can compare objects only in SQL statements and only for equality or inequality. Two objects of the same type are equal only if the values of their corresponding attributes are equal.

Like packaged subprograms, methods of the same kind (functions or procedures) can be overloaded. That is, you can use the same name for different methods if their formal parameters differ in number, order, or datatype family.

Every object type has a constructor method (constructor for short), which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type. PL/SQL never calls a constructor implicitly, so you must call it explicitly. Constructor calls are allowed wherever function calls are allowed.
Examples

In the SQL*Plus script below, an object type for a stack is defined. The last item added to a stack is the first item removed. The operations `push` and `pop` update the stack while preserving last in, first out (LIFO) behavior. The simplest implementation of a stack uses an integer array. Integers are stored in array elements, with one end of the array representing the top of the stack.

```sql
CREATE TYPE IntArray AS VARRAY(25) OF INTEGER;
CREATE TYPE Stack AS OBJECT (
    max_size INTEGER,
    top      INTEGER,
    position IntArray,
    MEMBER PROCEDURE initialize,
    MEMBER FUNCTION full RETURN BOOLEAN,
    MEMBER FUNCTION empty RETURN BOOLEAN,
    MEMBER PROCEDURE push (n IN INTEGER),
    MEMBER PROCEDURE pop (n OUT INTEGER)
);
CREATE TYPE BODY Stack AS
    MEMBER PROCEDURE initialize IS
        -- fill stack with nulls
        BEGIN
            top := 0;
            -- call constructor for varray and set element 1 to NULL
            position := IntArray(NULL);
            max_size := position.LIMIT;  -- use size constraint (25)
            position.EXTEND(max_size - 1, 1);  -- copy element 1
        END initialize;
    MEMBER FUNCTION full RETURN BOOLEAN IS
        -- return TRUE if stack is full
        BEGIN
            RETURN (top = max_size);
        END full;
    MEMBER FUNCTION empty RETURN BOOLEAN IS
        -- return TRUE if stack is empty
        BEGIN
            RETURN (top = 0);
        END empty;
```
MEMBER PROCEDURE push (n IN INTEGER) IS
  -- push integer onto stack
BEGIN
  IF NOT full THEN
    top := top + 1;
    position(top) := n;
  ELSE  -- stack is full
    RAISE_APPLICATION_ERROR(-20101, 'stack overflow');
  END IF;
END push;

MEMBER PROCEDURE pop (n OUT INTEGER) IS
  -- pop integer off stack and return its value
BEGIN
  IF NOT empty THEN
    n := position(top);
    top := top - 1;
  ELSE  -- stack is empty
    RAISE_APPLICATION_ERROR(-20102, 'stack underflow');
  END IF;
END pop;
END;

In methods push and pop, the built-in procedure raise_application_error issues user-defined error messages. That way, you can report errors to the client program and avoid returning unhandled exceptions to the host environment. In an object type, methods can reference attributes and other methods without a qualifier, as the following example shows:

CREATE TYPE Stack AS OBJECT (   top INTEGER,
  MEMBER FUNCTION full RETURN BOOLEAN,
  MEMBER PROCEDURE push (n IN INTEGER),
  ...
);

CREATE TYPE BODY Stack AS
...
  MEMBER PROCEDURE push (n IN INTEGER) IS
BEGIN
  IF NOT full THEN
    top := top + 1;
  ...
END push;
END;

The following example shows that you can nest object types:

```sql
CREATE TYPE Address AS OBJECT (
    street_address VARCHAR2(35),
    city VARCHAR2(15),
    state CHAR(2),
    zip_code INTEGER
);

CREATE TYPE Person AS OBJECT (
    first_name VARCHAR2(15),
    last_name VARCHAR2(15),
    birthday DATE,
    home_address Address,  -- nested object type
    phone_number VARCHAR2(15),
    ss_number INTEGER,
);
```

**Related Topics**

Functions, Packages, Procedures
OPEN Statement

The `OPEN` statement executes the multi-row query associated with an explicit cursor. It also allocates resources used by Oracle to process the query and identifies the result set, which consists of all rows that meet the query search criteria. The cursor is positioned before the first row in the result set. For more information, see "Managing Cursors" on page 6-6.

Syntax

```
open_statement

OPEN cursor_name (cursor_parameter_name);
```

Keyword and Parameter Description

**cursor_name**
This identifies an explicit cursor previously declared within the current scope and not currently open.

**cursor_parameter_name**
This identifies a cursor parameter; that is, a variable declared as the formal parameter of a cursor. (For the syntax of `cursor_parameter_declaration`, see "Cursors" 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 (thereby creating an implicit cursor) only the first time the statement is executed. All the parsed SQL statements are cached. A SQL statement must be reparsed only if it is aged out of the cache by a new SQL statement.

So, 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. Rather, 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. Therefore, they cannot return values to actual parameters. The values of actual parameters are used when the cursor is opened. The datatypes of the formal and actual parameters must be compatible. The query can also reference PL/SQL variables declared within its scope.

Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. Formal parameters declared with a default value need not have a corresponding actual parameter. They can simply 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**

Given the cursor declaration

```sql
CURSOR parts_cur IS SELECT part_num, part_price FROM parts;
```

the following statement opens the cursor:

```sql
OPEN parts_cur;
```

Given the cursor declaration

```sql
CURSOR emp_cur(my_ename VARCHAR2, my_comm NUMBER DEFAULT 0) IS SELECT * FROM emp WHERE ...;
```

any of the following statements opens the cursor:

```sql
OPEN emp_cur('LEE');
OPEN emp_cur('BLAKE', 300);
OPEN emp_cur(employee_name, 150);
```

**Related Topics**

CLOSE Statement, Cursors, FETCH Statement, LOOP Statements
OPEN-FOR Statement

The OPEN-FOR statement executes the multi-row query associated with a cursor variable. It also allocates resources used by Oracle to process the query and identifies the result set, which consists of all rows that meet the query search criteria. The cursor variable is positioned before the first row in the result set. For more information, see "Using Cursor Variables" on page 6-16.

Syntax

```
open_for_statement
```

```
OPEN cursor_variable_name:
:host_cursor_variable_name FOR select_statement;
```

Keyword and Parameter Description

**cursor_variable_name**

This identifies a cursor variable (or parameter) previously declared within the current scope.

**host_cursor_variable_name**

This identifies a cursor variable previously declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

**select_statement**

This is a query associated with cursor_variable, which returns a set of values. The query can reference bind variables and PL/SQL variables, parameters, and functions. The syntax of select_statement is similar to the syntax for select_into_statement defined in "SELECT INTO Statement" on page 13-164, except that select_statement cannot have an INTO clause.
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 variable 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:

```plsql
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM emp;
  OPEN :dept_cv FOR SELECT * FROM dept;
  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. No flexibility is lost, however, because you can pass whole queries (not just parameters) to a cursor variable.

You can pass a cursor variable to PL/SQL by calling a stored procedure that declares a cursor variable as one of its formal parameters. However, remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use a remote procedure call (RPC) to open a cursor variable.

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

To centralize data retrieval, you can group type-compatible queries in a stored procedure. When called, the following packaged procedure opens the cursor variable `emp_cv` for the chosen query:

```plsql
CREATE PACKAGE emp_data AS
  TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice IN INT);
END emp_data;
```
CREATE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice IN INT) IS
    BEGIN
      IF choice = 1 THEN
        OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
      ELSIF choice = 2 THEN
        OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
      ELSIF choice = 3 THEN
        OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
      END IF;
    END;
END emp_data;

For more flexibility, you can pass a cursor variable and a selector to a stored procedure that executes queries with different return types. Here is an example:

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 emp;
      ELSIF choice = 2 THEN
        OPEN generic_cv FOR SELECT * FROM dept;
      ELSIF choice = 3 THEN
        OPEN generic_cv FOR SELECT * FROM salgrade;
      END IF;
    END;
END admin_data;

**Related Topics**

CLOSE Statement, Cursor Variables, FETCH Statement, LOOP Statements
The **OPEN-FOR-USING** statement associates a cursor variable with a multi-row query, executes the query, identifies the result set, positions the cursor before the first row in the result set, then zeroes the rows-processed count kept by `%ROWCOUNT`. For more information, see "Using the OPEN-FOR, FETCH, and CLOSE Statements" on page 11-8.

**Syntax**

```
open_for_using_statement
```

- **OPEN**
  - `cursor_variable_name`
  - `host_cursor_variable_name`

- **FOR**
  - `dynamic_string`

- **USING**
  - `bind_argument`

**Keyword and Parameter Description**

- **cursor_variable_name**
  - This identifies a weakly typed cursor variable (one without a return type) previously declared within the current scope.

- **bind_argument**
  - This is an expression whose value is passed to the dynamic `SELECT` statement.

- **dynamic_string**
  - This is a string literal, variable, or expression that represents a multi-row `SELECT` statement.

- **host_cursor_variable_name**
  - This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible
with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

**USING ...**

This optional clause specifies a list of bind arguments. At run time, bind arguments in the **USING** clause replace corresponding placeholders in the dynamic **SELECT** statement.

**Usage Notes**

You use three statements to process a dynamic multi-row query: **OPEN-FOR-USING**, **FETCH**, and **CLOSE**. First, you **OPEN** a cursor variable **FOR** a multi-row query. Then, you **FETCH** rows from the result set. When all the rows are processed, you **CLOSE** the cursor variable.

The dynamic string can contain any multi-row **SELECT** statement (*without* the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement.

Every placeholder in the dynamic string must be associated with a bind argument in the **USING** clause. Numeric, character, and string literals are allowed in the **USING** clause, but Boolean literals (TRUE, FALSE, NULL) are not. To pass nulls to the dynamic string, you must use a workaround. See "Passing Nulls" on page 11-17.

Any bind arguments in the query are evaluated only when the cursor variable is opened. So, to fetch from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.

Dynamic SQL supports all the SQL datatypes. For example, bind arguments can be collections, LOBs, instances of an object type, and refs. As a rule, dynamic SQL does not support PL/SQL-specific types. For instance, bind arguments cannot be Booleans or index-by tables.
Example

In the following example, we declare a cursor variable, then associate it with a dynamic SELECT statement that returns rows from the emp table:

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;  -- define weak REF CURSOR type
    emp_cv   EmpCurTyp;  -- declare cursor variable
    my_ename VARCHAR2(15);
    my_sal   NUMBER := 1000;
BEGIN
    OPEN emp_cv FOR  -- open cursor variable
        'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
    ...  
END;
```

Related Topics

EXECUTE IMMEDIATE Statement
A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages have two parts: a specification (spec for short) and a body. For more information, see Chapter 9.

Syntax

```
package_declaration | package_spec

CREATE  OR REPLACE  PACKAGE  schema_name  package_name

AUTHID  CURRENT_USER  DEFINER

IS

PRAGMA  SERIALLY_REUSABLE

AS

collection_type_definition

record_type_definition

subtype_definition

collection_declaration

constant_declaration

exception_declaration

object_declaration

record_declaration

variable_declaration

cursor_spec

function_spec

procedure_spec

call spec

pragma_restrict_refs

END  package_name
```
Keyword and Parameter Description

**AUTHID**

This determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights Versus Definer Rights" on page 8-47.
call_spec
This publishes a Java method or external C function in the Oracle data dictionary. It publishes the routine by mapping its name, parameter types, and return type to their SQL counterparts. For more information, see Oracle9i Java Stored Procedures Developer’s Guide and/or Oracle9i Application Developer’s Guide - Fundamentals.

collection_declaration
This declares a collection (nested table, index-by table, or varray). For the syntax of collection_declaration, see "Collections" on page 13-28.

collection_type_definition
This defines a collection type using the datatype specifier TABLE or VARRAY.

constant_declaration
This declares a constant. For the syntax of constant_declaration, see "Constants and Variables" on page 13-38.

cursor_body
This defines the underlying implementation of an explicit cursor. For the syntax of cursor_body, see "Cursors" on page 13-53.

cursor_spec
This declares the interface to an explicit cursor. For the syntax of cursor_spec, see "Cursors" on page 13-53.

exception_declaration
This declares an exception. For the syntax of exception_declaration, see "Exceptions" on page 13-63.

function_body
This defines the underlying implementation of a function. For the syntax of function_body, see "Functions" on page 13-90.

function_spec
This declares the interface to a function. For the syntax of function_spec, see "Functions" on page 13-90.
object_declaration
This declares an object (instance of an object type). For the syntax of object_declaration, see "Object Types" on page 13-120.

package_name
This identifies a package stored in the database. For naming conventions, see "Identifiers" on page 2-4.

pragma_restrict_refs
This is pragma RESTRICT_REFERENCES, which lets you check for violations of "purity" rules. To be callable from SQL statements, a function must obey those rules, which are meant to control side effects. If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed). For the syntax of the pragma, see "RESTRICT_REFERENCES Pragma" on page 13-154.

The pragma asserts that a function does not read and/or write database tables and/or package variables. For more information about the purity rules and pragma RESTRICT_REFERENCES, see Oracle9i Application Developer’s Guide - Fundamentals.

PRAGMA SERIALLY_REUSABLE
This pragma lets you mark a package as serially reusable. You can so mark a package if its state is needed only for the duration of one call to the server (for example, an OCI call to the server or a server-to-server RPC). For more information, see Oracle9i Application Developer’s Guide - Fundamentals.

procedure_body
This defines the underlying implementation of a procedure. For the syntax of procedure_body, see "Procedures" on page 13-143.

procedure_spec
This declares the interface to a procedure. For the syntax of procedure_spec, see "Procedures" on page 13-143.

record_declaration
This declares a user-defined record. For the syntax of record_declaration, see "Records" on page 13-150.
record_type_definition
This defines a record type using the datatype specifier RECORD or the attribute %ROWTYPE.

schema_name
This qualifier identifies the schema containing the package. If you omit schema_name, Oracle assumes the package is in your schema.

variable_declaration
This declares a variable. For the syntax of variable_declaration, see "Constants and Variables" on page 13-38.

Usage Notes
You cannot define packages in a PL/SQL block or subprogram. However, you can use any Oracle tool that supports PL/SQL to create and store packages in an Oracle database. You can issue the CREATE PACKAGE and CREATE PACKAGE BODY statements from an Oracle Precompiler or OCI host program, or interactively from SQL*Plus.

Most packages have a spec and a body. The spec is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the spec.

Only subprograms and cursors have an underlying implementation (definition). So, if a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. However, the body can still be used to initialize items declared in the spec, as the following example shows:

```
CREATE PACKAGE emp_actions AS
  ...
  number_hired INTEGER;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
BEGIN
  number_hired := 0;
END emp_actions;
```
You can code and compile a spec without its body. Once the spec has been compiled, 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. Furthermore, you can debug, enhance, or replace a package body without changing the interface (package spec) to the package body. So, you need not recompile calling programs.

Cursors and subprograms declared in a package spec must be defined in the package body. Other program items declared in the package spec cannot be redeclared in the package body.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. So, except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception.

Related Topics

Collections, Cursors, Exceptions, Functions, Procedures, Records
A procedure is a subprogram that can take parameters and be invoked. Generally, you use a procedure to perform an action. A procedure has two parts: the specification and the body. The specification (spec for short) begins with the keyword **PROCEDURE** and ends with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses. The procedure body begins with the keyword **IS** (or **AS**) and ends with the keyword **END** followed by an optional procedure name.

The procedure body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. These items are local and cease to exist when you exit the procedure. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains handlers that deal with exceptions raised during execution. For more information, see “Understanding PL/SQL Procedures” on page 8-3.

**Syntax**

```
procedure_spec

PROCEDURE procedure_name parameter_declaration;
```

```
procedure_declaration | procedure body

CREATE OR REPLACE PROCEDURE procedure_name parameter_declaration AUTHID CURRENT_USER DEFINER IS AS
```
Keyword and Parameter Description

**AUTHID**
This clause determines whether a stored procedure executes with the privileges of its owner (the default) or current user and whether its unqualified references to schema objects are resolved in the schema of the owner or current user. You can override the default behavior by specifying CURRENT_USER. For more information, see "Invoker Rights Versus Definer Rights" on page 8-47.

**CREATE**
The optional CREATE clause lets you create standalone procedures, which are stored in the Oracle database. You can execute the CREATE statement interactively from SQL*Plus or from a program using native dynamic SQL.
**datatype**
This is a type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 13-38.

**exception_handler**
This associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of `exception_handler`, see "Exceptions" on page 13-63.

**expression**
This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

**function_declaration**
This declares a function. For the syntax of `function_declaration`, see "Functions" on page 13-90.

**IN, OUT, IN OUT**
These parameter modes define the behavior of formal parameters. An IN parameter lets you pass values to the subprogram being called. An OUT parameter lets you return values to the caller of the subprogram. An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

**item_declaration**
This declares a program object. For the syntax of `item_declaration`, see "Blocks" on page 13-11.

**NOCOPY**
This is a compiler hint (not directive), which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference instead of by value (the default). For more information, see "Passing Large Data Structures with the NOCOPY Compiler Hint" on page 8-17.
**parameter_name**
This identifies a formal parameter, which is a variable declared in a procedure spec and referenced in the procedure body.

**PRAGMA AUTONOMOUS_TRANSACTION**
This pragma instructs the PL/SQL compiler to mark a function as autonomous (independent). An autonomous transaction is an independent transaction started by another transaction, the main transaction. Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction. For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-53.

**procedure_name**
This identifies a user-defined procedure.

**type_definition**
This specifies a user-defined datatype. For the syntax of type_definition, see "Blocks" on page 13-11.

:= | DEFAULT
This operator or keyword lets you initialize IN parameters to default values.

**Usage Notes**
A procedure is called as a PL/SQL statement. For example, the procedure raise_salary might be called as follows:

```sql
raise_salary(emp_num, amount);
```

Inside a procedure, an IN parameter acts like a constant. So, you cannot assign it a value. An OUT parameter acts like a local variable. So, you can change its value and reference the value in any way. An IN OUT parameter acts like an initialized variable. So, you can assign it a value, which can be assigned to another variable. For summary information about the parameter modes, see Table 8–1 on page 8-16.

Unlike OUT and IN OUT parameters, IN parameters can be initialized to default values. For more information, see "Using Default Values for Subprogram Parameters" on page 8-19.

Before exiting a procedure, explicitly assign values to all OUT formal parameters. An OUT actual parameter can have a value before the subprogram is called. However,
when you call the subprogram, the value is lost unless you specify the compiler hint `NOCOPY` or the subprogram exits with an unhandled exception.

You can write the procedure spec and body as a unit. Or, you can separate the procedure spec from its body. That way, you can hide implementation details by placing the procedure in a package. You can define procedures in a package body without declaring their specs in the package spec. However, such procedures can be called only from inside the package.

At least one statement must appear in the executable part of a procedure. The `NULL` statement meets this requirement.

**Examples**

The following procedure debits a bank account:

```plsql
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
    old_balance REAL;
    new_balance REAL;
    overdrawn   EXCEPTION;
BEGIN
    SELECT bal INTO old_balance FROM accts WHERE acctno = acct_id;
    new_balance := old_balance - amount;
    IF new_balance < 0 THEN
        RAISE overdrawn;
    ELSE
        UPDATE accts SET bal = new_balance WHERE acctno = acct_id;
    END IF;
EXCEPTION
    WHEN overdrawn THEN
        ...
END debit_account;
```

In the following example, you call the procedure using named notation:

```plsql
debit_account(amount => 500, acct_id => 10261);
```

**Related Topics**

Collection Methods, Functions, Packages
RAISE Statement

The RAISE statement stops normal execution of a PL/SQL block or subprogram and transfers control to the appropriate exception handler. Normally, predefined exceptions are raised implicitly by the runtime system. However, RAISE statements can also raise predefined exceptions. User-defined exceptions must be raised explicitly by RAISE statements. For more information, see "Defining Your Own PL/SQL Exceptions" on page 7-7.

Syntax

```
raise statement
```

Keyword and Parameter Description

```
exception_name
```

This identifies a predefined or user-defined exception. For a list of the predefined exceptions, see "Predefined PL/SQL Exceptions" on page 7-4.

Usage Notes

PL/SQL blocks and subprograms should RAISE an exception only when an error makes it undesirable or impossible 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. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. In the latter case, PL/SQL returns an *unhandled exception* error to the host environment.

Omitting the exception name in a RAISE statement, which is allowed only in an exception handler, reraises the current exception. When a parameterless RAISE statement executes in an exception handler, the first block searched is the enclosing block, not the current block.
Example

In the following example, you raise an exception when an inventoried part is out of stock:

IF quantity_on_hand = 0 THEN
   RAISE out_of_stock;
END IF;

Related Topics

Exceptions
Records

Records are items of type RECORD. Records have uniquely named fields that can store data values of different types. Thus, a record lets you treat related but dissimilar data as a logical unit. For more information, see "What Is a Record?" on page 5-46.

Syntax

```
TYPE type_name IS RECORD (
  field_declaration
);
```

Keyword and Parameter Description

**datatype**

This is a datatype specifier. For the syntax of datatype, see "Constants and Variables" on page 13-38.

**expression**

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of expression, see "Expressions" on page 13-72. When the declaration is elaborated, the value of expression is assigned to the field. The value and the field must have compatible datatypes.
field_name
This identifies a field in a user-defined record.

NOT NULL
This constraint prevents the assigning of nulls to a field. 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
This identifies a user-defined record.

type_name
This identifies a user-defined record type that was defined using the datatype specifier RECORD.

:= | DEFAULT
This operator or keyword lets you initialize 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. Also, a record can be initialized in its declaration, as the following example shows:

DECLARE
    TYPE TimeTyp IS RECORD(
        seconds SMALLINT := 0,
        minutes SMALLINT := 0,
        hours   SMALLINT := 0);

The next example shows that you can use the %TYPE attribute to specify the datatype of a field. It also shows that you can add the NOT NULL constraint to any field declaration and thereby prevent the assigning of nulls to that field. Fields declared as NOT NULL must be initialized.

DECLARE
    TYPE DeptRecTyp IS RECORD(
        deptno NUMBER(2) NOT NULL := 99,
        dname  dept.dname%TYPE,
        loc    dept.loc%TYPE);
    dept_rec DeptRecTyp;
To reference individual fields in a record, you use dot notation. For example, you might assign a value to the field `dname` in the record `dept_rec` as follows:

```plsql
dept_rec.dname := 'PURCHASING';
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once. This can be done in two ways. First, you can assign one user-defined record to another if they have the same datatype. (Having fields that match exactly is not enough.) You can assign a `%ROWTYPE` record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes.

Second, 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.

You can declare and reference nested records. That is, a record can be the component of another record, as the following example shows:

```plsql
DECLARE
  TYPE TimeTyp IS RECORD(
    minutes SMALLINT,
    hours   SMALLINT);
  TYPE MeetingTyp IS RECORD(
    day     DATE,
    time_of TimeTyp,   -- nested record
    place   VARCHAR2(20),
    purpose VARCHAR2(50));
  TYPE PartyTyp IS RECORD(
    day     DATE,
    time_of TimeTyp,   -- nested record
    place   VARCHAR2(15));
  meeting MeetingTyp;
  seminar MeetingTyp;
  party   PartyTyp;

  seminar.time_of := meeting.time_of;
```

Such assignments are allowed even if the containing records have different datatypes.
User-defined records follow the usual scoping and instantiation rules. In a package, they are instantiated when you first reference the package and cease to exist when you end the database session. In a block or subprogram, they are instantiated when you enter the block or subprogram and cease to exist when you exit the block or subprogram.

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions. The restrictions that apply to scalar parameters also apply to user-defined records.

You can specify a RECORD type in the RETURN clause of a function spec. That allows the function to return a user-defined record of the same type. When calling 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
```

**Example**

In the following example, you define a RECORD type named DeptRecTyp, declare a record named dept_rec, then select a row of values into the record:

```
DECLARE
  TYPE DeptRecTyp IS RECORD(
    deptno NUMBER(2),
    dname VARCHAR2(14),
    loc VARCHAR2(13));
  dept_rec DeptRecTyp;
  ...
BEGIN
  SELECT deptno, dname, loc INTO dept_rec FROM dept
  WHERE deptno = 20;
```

**Related Topics**

Collections, Functions, Procedures, Packages
To be callable from SQL statements, a stored function must obey certain “purity” rules, which are meant to control side effects. (See “Controlling Side Effects of PL/SQL Subprograms" on page 8-9.) 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 violations of the rules, you can use the pragma (compiler directive) `RESTRICT_REFERENCES`. The pragma asserts that a function does not read and/or write database tables and/or package variables. For more information, see Oracle9i Application Developer’s Guide - Fundamentals.

Syntax

```
PRAGMA RESTRICT_REFERENCES
```

Keyword and Parameter Description

**DEFAULT**
This specifies that the pragma applies to all functions in the package spec or object type spec. You can still declare the pragma for individual functions. Such pragmas override the default pragma.

**function_name**
This identifies a user-defined function.

**PRAGMA**
This keyword signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler.
RESTRICT_REFERENCES Pragma

RNDS
This asserts that the function reads no database state (does not query database tables).

RNPS
This asserts that the function reads no package state (does not reference the values of packaged variables)

TRUST
This asserts that the function can be trusted not to violate one or more rules.

WNDS
This asserts that the function writes no database state (does not modify database tables).

WNPS
This asserts that the function writes no package state (does not change the values of packaged variables).

Usage Notes
You can declare the pragma RESTRICT_REFERENCES only in a package spec or object type spec. You can specify up to four constraints (RNDS, RNPS, WNDS, WNPS) in any order. To call the function from parallel queries, you must specify all four constraints. No constraint implies another. For example, WNPS does not imply RNPS.

When you specify TRUST, the function body is not checked for violations of the constraints listed in the pragma. The function is trusted not to violate them.

If you specify DEFAULT instead of a function name, the pragma applies to all functions in the package spec or object type spec (including, in the latter case, the system-defined constructor). You can still declare the pragma for individual functions. Such pragmas override the default pragma.

A RESTRICT_REFERENCES pragma can apply to only one function declaration. So, a pragma that references the name of overloaded functions always applies to the nearest foregoing function declaration.
The following pragma asserts that packaged function `balance` writes no database state (WNDS) and reads no package state (RNPS):

```sql
CREATE PACKAGE loans AS
  ...
  FUNCTION balance RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES (balance, WNDS, RNPS);
END loans;
```

A pragma that references the name of overloaded functions always applies to the nearest foregoing function declaration. So, in the following example, the pragma applies to the second declaration of `credit_ok`:

```sql
CREATE PACKAGE loans AS
  FUNCTION credit_ok (amount_limit NUMBER) RETURN BOOLEAN;
  FUNCTION credit_ok (time_limit DATE) RETURN BOOLEAN;
  PRAGMA RESTRICT_REFERENCES (credit_ok, WNDS);
  ...
END loans;
```

Related Topics

- AUTONOMOUS_TRANSACTION Pragma
- EXCEPTION_INIT Pragma
- SERIALLY_REUSABLE Pragma
RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. In a function, the RETURN statement also sets the function identifier to the return value. For more information, see "Using the RETURN Statement" on page 8-8.

Syntax

```
RETURN (expression);
```

Keyword and Parameter Description

**expression**

This is an arbitrarily complex 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

Do not confuse the RETURN statement with the RETURN clause in a function spec, which specifies the datatype of the return value.

A subprogram can contain several RETURN statements, none of which need be the last lexical statement. Executing any of them completes the subprogram immediately. However, to have multiple exit points in a subprogram is a poor programming practice.

In procedures, a RETURN statement cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.
However, 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. Therefore, 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.

The RETURN statement can also be used in an anonymous block to exit the block (and all enclosing blocks) immediately, but the RETURN statement cannot contain an expression.

Example

In the following example, the function balance returns the balance of a specified bank account:

```sql
FUNCTION balance (acct_id INTEGER) RETURN REAL IS
    acct_bal  REAL;
BEGIN
    SELECT bal INTO acct_bal FROM accts WHERE acctno = acct_id;
    RETURN acct_bal;
END balance;
```

Related Topics

Functions, Procedures
ROLLBACK Statement

The ROLLBACK statement is the inverse of the COMMIT statement. It undoes some or all database changes made during the current transaction. For more information, see "Overview of Transaction Processing in PL/SQL" on page 6-43.

Syntax

```
rollback_statement
```

Keyword and Parameter Description

**ROLLBACK**
When a parameterless ROLLBACK statement is executed, all database changes made during the current transaction are undone.

**ROLLBACK TO**
This statement undoes all database changes (and releases all locks acquired) since the savepoint identified by *savepoint_name* was marked.

**SAVEPOINT**
This keyword is optional and has no effect except to improve readability.

**savepoint_name**
This is an undeclared identifier, which marks the current point in the processing of a transaction. For naming conventions, see "Identifiers" on page 2-4.

**WORK**
This keyword is optional and has no effect except to improve readability.
Usage Notes

All savepoints marked after the savepoint to which you roll back are erased. However, the savepoint to which you roll back is not erased. For example, if you mark savepoints A, B, C, and D in that order, then roll back to savepoint B, only savepoints C and D are erased.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

In SQL, the FORCE clause manually rolls back an in-doubt distributed transaction. However, PL/SQL does not support this clause. For example, the following statement is illegal:

ROLLBACK WORK FORCE '24.37.85'; -- illegal

In embedded SQL, the RELEASE option frees all Oracle resources (locks and cursors) held by a program and disconnects from the database. However, PL/SQL does not support this option. For example, the following statement is illegal:

ROLLBACK WORK RELEASE; -- illegal

Related Topics

COMMIT Statement, SAVEPOINT Statement
The `%ROWTYPE` attribute provides a record type that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. Fields in a record and corresponding columns in a row have the same names and datatypes.

You can use the `%ROWTYPE` attribute in variable declarations as a datatype specifier. Variables declared using `%ROWTYPE` are treated like those declared using a datatype name. For more information, see "Using `%ROWTYPE`" on page 2-14.

**Syntax**

```
rowtype_attribute
```

**Keyword and Parameter Description**

- `cursor_name`
  This identifies an explicit cursor previously declared within the current scope.

- `cursor_variable_name`
  This identifies a PL/SQL strongly (not weakly) typed cursor variable previously declared within the current scope.

- `table_name`
  This identifies a database table (or view) that must be accessible when the declaration is elaborated.
%ROWTYPE Attribute

Usage Notes

The %ROWTYPE attribute lets you declare records structured like a row of data in a database table. To reference a field in the record, you use dot notation. For example, you might reference the deptno field as follows:

IF emp_rec.deptno = 20 THEN ...

You can assign the value of an expression to a specific field, as follows:

emp_rec.sal := average * 1.15;

There are two ways to assign values to all fields in a record at once. First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor. Second, you can assign a list of column values to a record by using the SELECT or FETCH statement. The column names must appear in the order in which they were declared. Select-items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases.

Examples

In the example below, you use %ROWTYPE to declare two records. The first record stores a row selected from the emp table. The second record stores a row fetched from the c1 cursor.

DECLARE
  emp_rec emp%ROWTYPE;
  CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
  dept_rec c1%ROWTYPE;

In the next example, you select a row from the emp table into a %ROWTYPE record:

DECLARE
  emp_rec emp%ROWTYPE;
  ...
BEGIN
  SELECT * INTO emp_rec FROM emp WHERE empno = my_empno;
  IF (emp_rec.deptno = 20) AND (emp_rec.sal > 2000) THEN
    ...
  END IF;
END;

Related Topics

Constants and Variables, Cursors, Cursor Variables, FETCH Statement
SAVEPOINT Statement

The SAVEPOINT statement names and marks the current point in the processing of a transaction. With the ROLLBACK TO statement, savepoints let you undo parts of a transaction instead of the whole transaction. For more information, see "Overview of Transaction Processing in PL/SQL" on page 6-43.

Syntax

```
SAVEPOINT savepoint_name;
```

Keyword and Parameter Description

**savepoint_name**
This is an undeclared identifier, which marks the current point in the processing of a transaction.

Usage Notes

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. However, the savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints. Savepoint names can be reused within a transaction. This moves the savepoint from its old position to the current point in the transaction.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent. However, you can only roll back to the most recently marked savepoint.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

Related Topics

COMMIT Statement, ROLLBACK Statement
The \texttt{SELECT INTO} statement retrieves data from one or more database tables, then assigns the selected values to variables or fields. For a full description of the \texttt{SELECT} statement, see \textit{Oracle9i SQL Reference}.

\textbf{Syntax}

\texttt{SELECT}\hspace{1cm} select\_item, \hspace{1cm} select\_into\_statement\hspace{1cm} INTO\hspace{1cm} variable\_name, record\_name \hspace{1cm} BULK COLLECT INTO\hspace{1cm} collection\_name, host\_array\_name,\hspace{1cm} FROM\hspace{1cm} alias, rest\_of\_statement ;

\hspace{1cm} (subquery \hspace{1cm} TABLE (subquery2) table\_reference) \hspace{1cm} DISTINCT \hspace{1cm} UNIQUE \hspace{1cm} ALL
Keyword and Parameter Description

**alias**
This is another (usually short) name for the referenced column, table, or view.

**BULK COLLECT**
This clause 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. For more information, see "Reducing Loop Overhead for Collections with Bulk Binds" on page 5-32.

**collection_name**
This identifies a declared collection into which select_item values are bulk fetched. For each select_item, there must be a corresponding, type-compatible collection in the list.

**function_name**
This identifies a user-defined function.
**host_array_name**
This identifies an array (declared in a PL/SQL host environment and passed to PL/SQL as a bind variable) into which select_item values are bulk 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**
This is a literal that represents a number or a value implicitly convertible to a number.

**parameter_name**
This identifies a formal parameter of a user-defined function.

**record_name**
This identifies 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**
This is anything that can legally follow the FROM clause in a SELECT statement except the SAMPLE clause.

**schema_name**
This qualifier identifies the schema containing the table or view. If you omit schema_name, Oracle assumes the table or view is in your schema.

**subquery**
This is a SELECT statement that provides a set of rows for processing. Its syntax is like that of select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 13-164.

**table_reference**
This identifies a table or view that must be accessible when you execute the SELECT statement, and for which you must have SELECT privileges. For the syntax of table_reference, see "DELETE Statement" on page 13-57.
TABLE (subquery2)
The operand of TABLE is a SELECT statement that returns a single column value, which must be a nested table or a varray. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

variable_name
This identifies a previously declared variable into which a select_item value is fetched. For each select_item value returned by the query, there must be a corresponding, type-compatible variable in the list.

Usage Notes
The BULK COLLECT clause tells the SQL engine to bulk-bind output collections before returning them. It bulk-binds all collections referenced in the INTO list. The corresponding columns can store scalar or composite values including objects.

When you use a SELECT INTO statement without the BULK COLLECT clause, it should return only one row. If it returns more than one row, PL/SQL raises the predefined exception TOO_MANY_ROWS.

However, if no rows are returned, PL/SQL raises NO_DATA_FOUND unless the SELECT statement called a SQL aggregate function such as AVG or SUM. (SQL aggregate functions always return a value or a null. So, a SELECT INTO statement that calls an aggregate function never raises NO_DATA_FOUND.)

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of a SELECT INTO statement.
Examples

The following SELECT statement returns an employee’s name, job title, and salary from the emp database table:

```
SELECT ename, job, sal INTO my_ename, my_job, my_sal FROM emp
  WHERE empno = my_empno;
```

In the following example, the SQL engine loads the entire empno and ename database columns into nested tables before returning the tables to the PL/SQL:

```
DECLARE
  TYPE NumTab IS TABLE OF emp.empno%TYPE;
  TYPE NameTab IS TABLE OF emp.ename%TYPE;
  enums NumTab;  -- no need to initialize
  names NameTab;
BEGIN
  SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;
  ...;
END;
```

Related Topics

Assignment Statement, FETCH Statement, %ROWTYPE Attribute
SERIALLY_REUSABLE Pragma

The pragma `SERIALLY_REUSABLE` lets you mark a package as `serially reusable`. You can so mark a package if its state is needed only for the duration of one call to the server (for example, an OCI call to the server or a server-to-server RPC). For more information, see Oracle9i Application Developer's Guide - Fundamentals.

Syntax

```plsql
serially_reusablePragma
```

Keyword and Parameter Description

**PRAGMA**

This keyword signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at runtime. They do not affect the meaning of a program; they simply convey information to the compiler.

Usage Notes

You can mark a bodiless package as serially reusable. If a package has a spec and body, you must mark both. You cannot mark only the body.

The global memory for serially reusable packages is pooled in the System Global Area (SGA), not allocated to individual users in the User Global Area (UGA). That way, the package work area can be reused. When the call to the server ends, the memory is returned to the pool. Each time the package is reused, its public variables are initialized to their default values or to `NULL`.

Serially reusable packages cannot be accessed from database triggers. If you try, Oracle generates an error.
Examples

In the following example, you create 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, EXCEPTION_INIT Pragma,
RESTRICT_REFERENCES Pragma
SET TRANSACTION Statement

The `SET TRANSACTION` statement begins a read-only or read-write transaction, establishes an isolation level, or assigns the current transaction to a specified rollback segment. Read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables. For more information, see "Setting Transaction Properties with SET TRANSACTION" on page 6-48.

Syntax

```
SET TRANSACTION
  READ ONLY
  READ WRITE
  ISOLATION LEVEL
    SERIALIZABLE
    READ COMMITTED
  USE ROLLBACK SEGMENT rollback_segment;
```

Keyword and Parameter Description

**READ ONLY**
This clause establishes the current transaction as read-only. If a transaction is set 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.

**READ WRITE**
This clause establishes the current transaction as read-write. The use of READ WRITE does not affect other users or transactions. If the transaction executes a data manipulation statement, Oracle assigns the transaction to a rollback segment.

**ISOLATION LEVEL**
This clause specifies how transactions that modify the database are handled. When you specify SERIALIZABLE, if a serializable transaction tries to execute a SQL data manipulation statement that modifies any table already modified by an uncommitted transaction, the statement fails.
To enable SERIALizable mode, your DBA must set the Oracle initialization parameter COMPATIBLE to 7.3.0 or higher. When you specify READ COMMITTED, if a transaction includes SQL data manipulation statements that require row locks held by another transaction, the statement waits until the row locks are released.

**USE ROLLBACK SEGMENT**

This clause assigns the current transaction to the specified rollback segment and establishes the transaction as read-write. You cannot use this parameter with the READ ONLY parameter in the same transaction because read-only transactions do not generate rollback information.

**NAME**

Lets you specify a name or some comment text for the transaction. This is better than using the COMMIT COMMENT feature because the name is available while the transaction is running, making it easier to monitor long-running and in-doubt transactions.

**Usage Notes**

The SET TRANSACTION statement must be the first SQL statement in your transaction and can appear only once in the transaction.

**Example**

In the following example, you establish a read-only transaction:

```sql
COMMIT;  -- end previous transaction
SET TRANSACTION READ ONLY;
SELECT ... FROM emp WHERE ...
SELECT ... FROM dept WHERE ...
SELECT ... FROM emp WHERE ...
COMMIT;  -- end read-only transaction
```

**Related Topics**

COMMIT Statement, ROLLBACK Statement, SAVEPOINT Statement
SQL Cursor

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicit cursor. PL/SQL lets you refer to the most recent implicit cursor as the SQL cursor, which always has these attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. They give you useful information about the execution of data manipulation statements. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. For more information, see "Managing Cursors" on page 6-6.

Syntax

```
sql_cursor
```

```
SQL %
    FOUND
    ISOPEN
    NOTFOUND
    ROWCOUNT
    BULK_ROWCOUNT(index)
```

Keyword and Parameter Description

**%BULK_ROWCOUNT**

This is a composite attribute designed for use with the FORALL statement. This attribute has the semantics of 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**

**%FOUND**

This attribute 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, it yields FALSE.
%ISOPEN
This attribute always yields FALSE because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%NOTFOUND
This attribute is the logical opposite of %FOUND. It yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it yields FALSE.

%ROWCOUNT
This attribute yields the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

SQL
This is the name of the Oracle implicit cursor.

Usage Notes
You can use cursor attributes in procedural statements but not in SQL statements. Before Oracle opens the SQL cursor automatically, the implicit cursor attributes yield 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. So, if you want to save an attribute value for later use, assign it to a Boolean 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. However, a SELECT INTO statement that calls a SQL aggregate function never raises NO_DATA_FOUND because those functions always return a value or a null. In such cases, SQL%NOTFOUND yields FALSE.

%BULK_ROWCOUNT is not maintained for bulk inserts because that would be redundant. For example, the FORALL statement below inserts one row per iteration. So, after each iteration, %BULK_ROWCOUNT would return 1:

FORALL i IN 1..15
    INSERT INTO emp (sal) VALUES (sals(i));
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.

%FOUND and %NOTFOUND refer only to the last execution of the SQL statement. However, you can use %BULK_ROWCOUNT to infer their values for individual executions. For example, when %BULK_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

**Examples**

In the following example, %NOTFOUND is used to insert a row if an update affects no rows:

```sql
UPDATE emp SET sal = sal * 1.05 WHERE empno = my_empno;
IF SQL%NOTFOUND THEN
    INSERT INTO emp VALUES (my_empno, my_ename, ...);
END IF;
```

In the next example, you use %ROWCOUNT to raise an exception if more than 100 rows are deleted:

```sql
DELETE FROM parts WHERE status = 'OBSOLETE';
IF SQL%ROWCOUNT > 100 THEN -- more than 100 rows were deleted
    RAISE large_deletion;
END IF;
```

Here is an example that uses %BULK_ROWCOUNT:

```sql
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20, 50);
BEGIN
    FORALL j IN depts.FIRST..depts.LAST
        UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
    IF SQL%BULK_ROWCOUNT(3) = 0 THEN
        ...
    END;
END;
```

**Related Topics**

Cursors, Cursor Attributes
The function `SQLCODE` returns the number code associated with the most recently raised exception. `SQLCODE` is meaningful only in an exception handler. Outside a handler, `SQLCODE` always returns 0.

For internal exceptions, `SQLCODE` returns the number of the associated Oracle error. The number that `SQLCODE` returns is negative unless the Oracle error is `no data found`, in which case `SQLCODE` returns `+100`.

For user-defined exceptions, `SQLCODE` returns `+1` unless you used the pragma `EXCEPTION_INIT` to associate the exception with an Oracle error number, in which case `SQLCODE` returns that error number. For more information, see "Retrieving the Error Code and Error Message: `SQLCODE` and `SQLERRM`" on page 7-18.

**Syntax**

```sql
sqlcode_function
-> SQLCODE
```

**Usage Notes**

`SQLCODE` is especially useful in the `OTHERS` exception handler because it lets you identify which internal exception was raised.

You cannot use `SQLCODE` directly in a SQL statement. First, you must assign the value of `SQLCODE` to a local variable, as follows:

```sql
my_sqlcode := SQLCODE;
...
INSERT INTO errors VALUES (my_sqlcode, ...);
```

When using pragma `RESTRICT_REFERENCES` to assert the purity of a stored function, you cannot specify the constraints `WNPS` and `RNPS` if the function calls `SQLCODE`.
Example

In the following example, you insert the value of SQLCODE into an audit table:

```plsql
DECLARE
    my_sqlcode NUMBER;
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        my_sqlcode := SQLCODE;
        INSERT INTO audits VALUES (my_sqlcode, ...);
END;
```

Related Topics

Exceptions, SQLERRM Function
The function `SQLERRM` returns the error message associated with its error-number argument or, if the argument is omitted, with the current value of `SQLCODE`.

`SQLERRM` with no argument is meaningful only in an exception handler. Outside a handler, `SQLERRM` with no argument always returns the message `normal, successful completion`.

For internal exceptions, `SQLERRM` returns the message associated with the Oracle error that occurred. The message begins with the Oracle error code.

For user-defined exceptions, `SQLERRM` returns the message `user-defined exception` unless you used the pragma `EXCEPTION_INIT` to associate the exception with an Oracle error number, in which case `SQLERRM` returns the corresponding error message. For more information, see "Retrieving the Error Code and Error Message: `SQLCODE` and `SQLERRM`" on page 7-18.

### Syntax

```
sqlerrm_function
```

### Keyword and Parameter Description

- **error_number**
  
  This must be a valid Oracle error number. For a list of Oracle errors, see Oracle9i Database Error Messages.

### Usage Notes

`SQLERRM` is especially useful in the `OTHERS` exception handler because it lets you identify which internal exception was raised.

You can pass an error number to `SQLERRM`, in which case `SQLERRM` returns the message associated with that error number. The error number passed to `SQLERRM` should be negative. Passing a zero to `SQLERRM` always returns the following message:

```
ORA-0000: normal, successful completion
```
Passing a positive number to SQLERRM always returns the message User-Defined Exception unless you pass +100, in which case SQLERRM returns the following message:
ORA-01403: no data found

You cannot use SQLERRM directly in a SQL statement. First, you must assign the value of SQLERRM to a local variable, as follows:

```sql
my_sqlerrm := SQLERRM;
```

```sql
... INSERT INTO errors VALUES (my_sqlerrm, ...);
```

When using pragma RESTRICT_REFERENCES to assert the purity of a stored function, you cannot specify the constraints WNPS and RNPS if the function calls SQLERRM.

**Example**

In the following example, the string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to `my_sqlerrm`:

```sql
DECLARE
    my_sqlerrm VARCHAR2(150);
...
BEGIN
...
EXCEPTION
...
    WHEN OTHERS THEN
        my_sqlerrm := SUBSTR(SQLERRM, 1, 150);
        INSERT INTO audits VALUES (my_sqlerrm, ...);
END;
```

**Related Topics**

Exceptions, SQLCODE Function
%TYPE Attribute

The %TYPE attribute provides the datatype of a field, record, nested table, database column, or variable. You can use the %TYPE attribute as a datatype specifier when declaring constants, variables, fields, and parameters. For more information, see "Using %TYPE" on page 2-13.

Syntax

type_attribute

collection_name

cursor_variable_name

object_name

record_name

field_name

db_table_name

column_name

variable_name

Keyword and Parameter Description

**collection_name**
This identifies a nested table, index-by table, or varray previously declared within the current scope.

**cursor_variable_name**
This identifies 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**
This refers to a table and column that must be accessible when the declaration is elaborated.
object_name
This identifies an object (instance of an object type) previously declared within the current scope.

record_name
This identifies a user-defined or %ROWTYPE record previously declared within the current scope.

record_name.field_name
This identifies a field in a user-defined or %ROWTYPE record previously declared within the current scope.

variable_name
This identifies a variable previously declared in the same scope.

Usage Notes
The %TYPE attribute is particularly useful when declaring variables, fields, and parameters that refer to database columns. However, the NOT NULL column constraint is not inherited by items declared using %TYPE.

Related Topics
Constants and Variables, %ROWTYPE Attribute
The UPDATE statement changes the values of specified columns in one or more rows in a table or view. For a full description of the UPDATE statement, see Oracle9i SQL Reference.

**Syntax**

```
update_statement
  <- UPDATE
        (table_reference) alias
        
  <- SET
        (column_name) = (sql_expression)
        
  <- WHERE
        search_condition
        
        CURRENT OF cursor_name
        returning_clause
```

**Keyword and Parameter Description**

**alias**
This is another (usually short) name for the referenced table or view and is typically used in the WHERE clause.
**column_name**
This is the name of the column (or one of the columns) to be updated. It must be the name of a column in the referenced table or view. A column name cannot be repeated in the column_name list. Column names need not appear in the UPDATE statement in the same order that they appear in the table or view.

**returning_clause**
This clause lets you return values from updated rows, thereby eliminating the need to SELECT the rows afterward. You can retrieve the column values into variables and/or host variables, or into collections and/or host arrays. However, you cannot use the RETURNING clause for remote or parallel updates. For the syntax of returning_clause, see "DELETE Statement" on page 13-57.

**SET column_name = sql_expression**
This clause assigns the value of sql_expression to the column identified by column_name. If sql_expression contains references to columns in the table being updated, the references are resolved in the context of the current row. The old column values are used on the right side of the equal sign.

In the following example, you increase every employee’s salary by 10%. The original value of the sal column is multiplied by 1.10, then the result is assigned to the sal column overwriting the original value.

```
UPDATE emp SET sal = sal * 1.10;
```

**SET column_name = (subquery3)**
This clause assigns the value retrieved from the database by subquery3 to the column identified by column_name. The subquery must return exactly one row and one column.

**SET (column_name, column_name, ...) = (subquery4)**
This clause assigns the values retrieved from the database by subquery4 to the columns in the column_name list. The subquery must return exactly one row that includes all the columns listed.

The column values returned by the subquery are assigned to the columns in the column list in order. The first value is assigned to the first column in the list, the second value is assigned to the second column in the list, and so on.
In the following correlated query, the column item_id is assigned the value stored in item_num, and the column price is assigned the value stored in item_price:

```
UPDATE inventory inv -- alias
SET (item_id, price) =
    (SELECT item_num, item_price FROM item_table
     WHERE item_name = inv.item_name);
```

**sql_expression**
This is any valid SQL expression. For more information, see *Oracle9i SQL Reference*.

**subquery**
This is a `SELECT` statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See “SELECT INTO Statement” on page 13-164.

**table_reference**
This identifies a table or view that must be accessible when you execute the `UPDATE` statement, and for which you must have `UPDATE` privileges. For the syntax of `table_reference`, see “DELETE Statement” on page 13-57.

**TABLE (subquery2)**
The operand of `TABLE` is a `SELECT` statement that returns a single column value, which must be a nested table or a varray. Operator `TABLE` informs Oracle that the value is a collection, not a scalar value.

**WHERE CURRENT OF cursor_name**
This clause refers to the latest row processed by the `FETCH` statement associated with the cursor identified by `cursor_name`. The cursor must be `FOR UPDATE` and must be open and positioned on a row.

If the cursor is not open, the `CURRENT OF` clause causes an error. If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception `NO_DATA_FOUND`.

**WHERE search_condition**
This clause chooses which rows to update in the database table. Only rows that meet the search condition are updated. If you omit the search condition, all rows in the table are updated.
Usage Notes

You can use the UPDATE WHERE CURRENT OF statement after a fetch from an open
cursor (this includes implicit fetches executed in a cursor FOR loop), provided the
associated query is FOR UPDATE. This statement updates the current row, that is, the
one just fetched.

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND,
%ROWCOUNT, and %ISOPEN let you access useful information about the execution of
an UPDATE statement.

Examples

In the following example, a 10% raise is given to analysts in department 20:

UPDATE emp SET sal = sal * 1.10
    WHERE job = ‘ANALYST’ AND DEPTNO = 20;

In the next example, an employee named Ford is promoted to the position of
Analyst and her salary is raised by 15%:

UPDATE emp SET job = ‘ANALYST’, sal = sal * 1.15
    WHERE ename = ‘FORD’;

In the final example, values returned from an updated row are stored in variables:

UPDATE emp SET sal = sal + 500 WHERE ename = ‘MILLER’
    RETURNING sal, ename INTO my_sal, my_ename;

Related Topics

DELETE Statement, FETCH Statement
This appendix provides several PL/SQL programs to guide you in writing your own. The sample programs illustrate several important PL/SQL concepts and features.

This appendix discusses the following topics:

Running the Programs
Sample 1. FOR Loop
Sample 2. Cursors
Sample 3. Scoping
Sample 4. Batch Transaction Processing
Sample 5. Embedded PL/SQL
Sample 6. Calling a Stored Procedure
Running the Programs

All the sample programs in this appendix and several others throughout this guide are available online. So, they are preceded by the following comment:

```
-- available online in file '<filename>'
```

You can find the online files in the PL/SQL demo directory. For the location of the directory, see the Oracle installation or user’s guide for your system. Here is a list of the files and their locations in this guide:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Location in This Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>examp1</td>
<td>&quot;Understanding the Main Features of PL/SQL&quot; on page 1-2</td>
</tr>
<tr>
<td>examp2</td>
<td>&quot;Conditional Control&quot; on page 1-9</td>
</tr>
<tr>
<td>examp3</td>
<td>&quot;Iterative Control&quot; on page 1-10</td>
</tr>
<tr>
<td>examp4</td>
<td>&quot;Using Aliases&quot; on page 2-16</td>
</tr>
<tr>
<td>examp7</td>
<td>&quot;Using Cursor FOR Loops&quot; on page 6-13</td>
</tr>
<tr>
<td>examp8</td>
<td>&quot;Passing Parameters to a Cursor FOR Loop&quot; on page 6-15</td>
</tr>
<tr>
<td>examp5</td>
<td>&quot;Some Examples of Cursor Attributes&quot; on page 6-37</td>
</tr>
<tr>
<td>examp6</td>
<td>&quot;Some Examples of Cursor Attributes&quot; on page 6-37</td>
</tr>
<tr>
<td>examp11</td>
<td>&quot;Example&quot; on page 13-17</td>
</tr>
<tr>
<td>examp12</td>
<td>&quot;Examples&quot; on page 13-44</td>
</tr>
<tr>
<td>examp13</td>
<td>&quot;Examples&quot; on page 13-44</td>
</tr>
<tr>
<td>examp14</td>
<td>&quot;Examples&quot; on page 13-44</td>
</tr>
<tr>
<td>sample1</td>
<td>&quot;Sample 1. FOR Loop&quot; on page A-3</td>
</tr>
<tr>
<td>sample2</td>
<td>&quot;Sample 2. Cursors&quot; on page A-4</td>
</tr>
<tr>
<td>sample3</td>
<td>&quot;Sample 3. Scoping&quot; on page A-6</td>
</tr>
<tr>
<td>sample4</td>
<td>&quot;Sample 4. Batch Transaction Processing&quot; on page A-7</td>
</tr>
<tr>
<td>sample5</td>
<td>&quot;Sample 5. Embedded PL/SQL&quot; on page A-11</td>
</tr>
<tr>
<td>sample6</td>
<td>&quot;Sample 6. Calling a Stored Procedure&quot; on page A-15</td>
</tr>
</tbody>
</table>

You run some samples interactively from SQL*Plus, others from Pro*C programs. You can experiment with the samples from any Oracle account. However, the Pro*C examples expect you to use the scott/tiger account.
Before trying the samples, you must create some database tables, then load the tables with data. You do that by running two SQL*Plus scripts, exambld and examplod, which are supplied with PL/SQL. You can find these scripts in the PL/SQL demo directory.

The first script builds the database tables processed by the sample programs. The second script loads (or reloads) the database tables. To run the scripts, invoke SQL*Plus, then issue the following commands:

```
SQL> START exambld
...
SQL> START examplod
```

**Sample 1. FOR Loop**

The following example uses a simple FOR loop to insert ten rows into a database table. The values of a loop index, counter variable, and either of two character strings are inserted. Which string is inserted depends on the value of the loop index.

**Input Table**

Not applicable.

**PL/SQL Block**

```
-- available online in file 'sample1'
DECLARE
  x NUMBER := 100;
BEGIN
  FOR i IN 1..10 LOOP
    IF MOD(i,2) = 0 THEN  -- i is even
      INSERT INTO temp VALUES (i, x, 'i is even');
    ELSE
      INSERT INTO temp VALUES (i, x, 'i is odd');
    END IF;
    x := x + 100;
  END LOOP;
  COMMIT;
END;
```
Output Table

SQL> SELECT * FROM temp ORDER BY col1;

<table>
<thead>
<tr>
<th>NUM_COL1</th>
<th>NUM_COL2</th>
<th>CHAR_COL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>i is odd</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>i is even</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>i is odd</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>i is even</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>i is odd</td>
</tr>
<tr>
<td>6</td>
<td>600</td>
<td>i is even</td>
</tr>
<tr>
<td>7</td>
<td>700</td>
<td>i is odd</td>
</tr>
<tr>
<td>8</td>
<td>800</td>
<td>i is even</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>i is odd</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>i is even</td>
</tr>
</tbody>
</table>

Sample 2. Cursors

The following example uses a cursor to select the five highest paid employees from the emp table.

Input Table

SQL> SELECT ename, empno, sal FROM emp ORDER BY sal DESC;

<table>
<thead>
<tr>
<th>ENAME</th>
<th>EMPNO</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KING</td>
<td>7839</td>
<td>5000</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7788</td>
<td>3000</td>
</tr>
<tr>
<td>FORD</td>
<td>7902</td>
<td>3000</td>
</tr>
<tr>
<td>JONES</td>
<td>7566</td>
<td>2975</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7698</td>
<td>2850</td>
</tr>
<tr>
<td>CLARK</td>
<td>7782</td>
<td>2450</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7499</td>
<td>1600</td>
</tr>
<tr>
<td>TURNER</td>
<td>7844</td>
<td>1500</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
<td>1300</td>
</tr>
<tr>
<td>WARD</td>
<td>7521</td>
<td>1250</td>
</tr>
<tr>
<td>MARTIN</td>
<td>7654</td>
<td>1250</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>1100</td>
</tr>
<tr>
<td>JAMES</td>
<td>7900</td>
<td>950</td>
</tr>
<tr>
<td>SMITH</td>
<td>7369</td>
<td>800</td>
</tr>
</tbody>
</table>
PL/SQL Block

```plsql
-- available online in file 'sample2'
DECLARE
    CURSOR c1 is
        SELECT ename, empno, sal FROM emp
        ORDER BY sal DESC; -- start with highest paid employee
    my_ename VARCHAR2(10);
    my_empno NUMBER(4);
    my_sal   NUMBER(7,2);
BEGIN
    OPEN c1;
    FOR i IN 1..5 LOOP
        FETCH c1 INTO my_ename, my_empno, my_sal;
        EXIT WHEN c1%NOTFOUND; /* in case the number requested */
        /* is more than the total */
        /* number of employees */
        INSERT INTO temp VALUES (my_sal, my_empno, my_ename);
        COMMIT;
    END LOOP;
    CLOSE c1;
END;
```

Output Table

```
SQL> SELECT * FROM temp ORDER BY col1 DESC;

<table>
<thead>
<tr>
<th>NUM_COL1</th>
<th>NUM_COL2</th>
<th>CHAR_COL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>3000</td>
<td>7902</td>
<td>FORD</td>
</tr>
<tr>
<td>3000</td>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>2975</td>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>2850</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
</tbody>
</table>
```
Sample 3. Scoping

The following example illustrates block structure and scope rules. An outer block declares two variables named x and counter and loops four times. Inside this loop is a sub-block that also declares a variable named x. The values inserted into the temp table show that the two x’s are indeed different.

Input Table

Not applicable.

PL/SQL Block

```sql
-- available online in file 'sample3'
DECLARE
  x NUMBER := 0;
  counter NUMBER := 0;
BEGIN
  FOR i IN 1..4 LOOP
    x := x + 1000;
    counter := counter + 1;
    INSERT INTO temp VALUES (x, counter, 'in OUTER loop');
    /* start an inner block */
    DECLARE
      x NUMBER := 0;  -- this is a local version of x
    BEGIN
      FOR i IN 1..4 LOOP
        x := x + 1;  -- this increments the local x
        counter := counter + 1;
        INSERT INTO temp VALUES (x, counter, 'inner loop');
      END LOOP;
    END;
  END Loop;
END;
```
Output Table

```
SQL> SELECT * FROM temp ORDER BY col2;

NUM_COL1  NUM_COL2  CHAR_COL
---------- ----------  -------------
1000       1          in OUTER loop
           2          inner loop
           3          inner loop
           4          inner loop
           5          inner loop
2000       6          in OUTER loop
           7          inner loop
           8          inner loop
           9          inner loop
           10         inner loop
3000       11         in OUTER loop
            12         inner loop
            13         inner loop
            14         inner loop
            15         inner loop
4000       16         in OUTER loop
            17         inner loop
            18         inner loop
            19         inner loop
            20         inner loop
```

Sample 4. Batch Transaction Processing

In the next example the accounts table is modified according to instructions stored in the action table. Each row in the action table contains an account number, an action to be taken (I, U, or D for insert, update, or delete), an amount by which to update the account, and a time tag used to sequence the transactions.

On an insert, if the account already exists, an update is done instead. On an update, if the account does not exist, it is created by an insert. On a delete, if the row does not exist, no action is taken.
Input Tables

SQL> SELECT * FROM accounts ORDER BY account_id;

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>BAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>6500</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
</tr>
</tbody>
</table>

SQL> SELECT * FROM action ORDER BY time_tag;

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>O</th>
<th>NEW_VALUE</th>
<th>STATUS</th>
<th>TIME_TAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>u</td>
<td>599</td>
<td></td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
<td>20099</td>
<td></td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td></td>
<td></td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>7</td>
<td>u</td>
<td>1599</td>
<td></td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
<td>399</td>
<td></td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>9</td>
<td>d</td>
<td></td>
<td></td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td></td>
<td></td>
<td>18-NOV-88</td>
</tr>
</tbody>
</table>

PL/SQL Block

-- available online in file 'sample4'
DECLARE
    CURSOR c1 IS
        SELECT account_id, oper_type, new_value FROM action
        ORDER BY time_tag
        FOR UPDATE OF status;
BEGIN
    FOR acct IN c1 LOOP  -- process each row one at a time
        acct.oper_type := upper(acct.oper_type);
        /*----------------------------------------*/
        /* Process an UPDATE. If the account to */
        /* be updated doesn’t exist, create a new */
        /* account.                               */
        /*----------------------------------------*/
        IF acct.oper_type = 'U' THEN
            UPDATE accounts SET bal = acct.new_value
                WHERE account_id = acct.account_id;
IF SQL%NOTFOUND THEN -- account didn’t exist. Create it.
    INSERT INTO accounts
    VALUES (acct.account_id, acct.new_value);
    UPDATE action SET status = 
    ’Update: ID not found. Value inserted.’
    WHERE CURRENT OF c1;
ELSE
    UPDATE action SET status = ’Update: Success.’
    WHERE CURRENT OF c1;
END IF;

/*================================================================================*/
/* Process an INSERT. If the account already */
/* exists, do an update of the account       */
/* instead.                                   */
/*================================================================================*/
ELSIF acct.oper_type = ’I’ THEN
BEGIN
    INSERT INTO accounts
    VALUES (acct.account_id, acct.new_value);
    UPDATE action set status = ’Insert: Success.’
    WHERE CURRENT OF c1;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN -- account already exists
        UPDATE accounts SET bal = acct.new_value
        WHERE account_id = acct.account_id;
        UPDATE action SET status = 
        ’Insert: Acct exists. Updated instead.’
        WHERE CURRENT OF c1;
END;

/*================================================================================*/
/* Process a DELETE. If the account doesn’t */
/* exist, set the status field to say that   */
/* the account wasn’t found.                 */
/*================================================================================*/
ELSIF acct.oper_type = ’D’ THEN
DELETE FROM accounts
    WHERE account_id = acct.account_id;

    IF SQL%NOTFOUND THEN -- account didn’t exist.
        UPDATE action SET status = ’Delete: ID not found.’
        WHERE CURRENT OF c1;
ELSE

Sample PL/SQL Programs  A-9
UPDATE action SET status = 'Delete: Success.'
WHERE CURRENT OF c1;
END IF;
/*--------------------------------------------*/
/* The requested operation is invalid. */
/*--------------------------------------------*/
ELSE  -- oper_type is invalid
UPDATE action SET status =
'Invalid operation. No action taken.'
WHERE CURRENT OF c1;
END IF;
END LOOP;
END LOOP;
COMMIT;
END;

Output Tables

SQL> SELECT * FROM accounts ORDER BY account_id;

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>BAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>399</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>599</td>
</tr>
<tr>
<td>4</td>
<td>6500</td>
</tr>
<tr>
<td>6</td>
<td>20099</td>
</tr>
<tr>
<td>7</td>
<td>1599</td>
</tr>
</tbody>
</table>

SQL> SELECT * FROM action ORDER BY time_tag;

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>O</th>
<th>NEW_VALUE</th>
<th>STATUS</th>
<th>TIME_TAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>u</td>
<td>599</td>
<td>Update: Success.</td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
<td>20099</td>
<td>Insert: Success.</td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td></td>
<td>Delete: Success.</td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>7</td>
<td>u</td>
<td>1599</td>
<td>Update: ID not found. Value inserted.</td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
<td>399</td>
<td>Insert: Acct exists. Updated instead.</td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>9</td>
<td>d</td>
<td></td>
<td>Delete: ID not found.</td>
<td>18-NOV-88</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td></td>
<td>Invalid operation. No action taken.</td>
<td>18-NOV-88</td>
</tr>
</tbody>
</table>
Sample 5. Embedded PL/SQL

The following example shows how you can embed PL/SQL in a high-level host language such as C and demonstrates how a banking debit transaction might be done.

Input Table

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID      BAL
---------- --------
1     1000
2     2000
3     1500
4     6500
5      500
```

PL/SQL Block in a C Program

```
/* available online in file 'sample5' */
#include <stdio.h>
char    buf[20];
EXEC SQL BEGIN DECLARE SECTION;
    int     acct;
    double  debit;
    double  new_bal;
    VARCHAR status[65];
    VARCHAR uid[20];
    VARCHAR pwd[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;

main()
{
    extern double atof();
    strcpy (uid.arr,"scott");
    uid.len=strlen(uid.arr);
    strcpy (pwd.arr,"tiger");
    pwd.len=strlen(pwd.arr);
```
printf("Embedded PL/SQL Debit Transaction Demo\n\n");
printf("Trying to connect...");
EXEC SQL WHENEVER SQLERROR GOTO errprint;
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
printf(" connected.\n");
for (;;)       /*  Loop infinitely */
{
    printf("Debit which account number? (-1 to end) ");
gets(buf);
    acct = atoi(buf);
    if (acct == -1)  /* Need to disconnect from Oracle */
        /* and exit loop if account is -1 */
        EXEC SQL COMMIT RELEASE;
        exit(0);
    printf("What is the debit amount? ");
gets(buf);
    debit = atof(buf);

/* ---------------------------------- */
/* ----- Begin the PL/SQL block ----- */
/* ---------------------------------- */
EXEC SQL EXECUTE
DECLARE
    insufficient_funds EXCEPTION;
    old_bal            NUMBER;
    min_bal            CONSTANT NUMBER := 500;
BEGIN
    SELECT bal INTO old_bal FROM accounts
        WHERE account_id = :acct;
    -- If the account doesn’t exist, the NO_DATA_FOUND
    -- exception will be automatically raised.
    :new_bal := old_bal - :debit;
    IF :new_bal >= min_bal THEN
        UPDATE accounts SET bal = :new_bal
            WHERE account_id = :acct;
        INSERT INTO journal
            VALUES (:acct, 'Debit', :debit, SYSDATE);
        :status := 'Transaction completed.';
    ELSE
        RAISE insufficient_funds;
    END IF;
    COMMIT;
}
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        :status := 'Account not found.';
        :new_bal := -1;
    WHEN insufficient_funds THEN
        :status := 'Insufficient funds.';
        :new_bal := old_bal;
    WHEN OTHERS THEN
        ROLLBACK;
        :status := 'Error: ' || SQLERRM(SQLCODE);
        :new_bal := -1;
END;
END-EXEC;

status.arr[status.len] = '\0';  /* null-terminate */
/* the string */
printf("\n\n   Status:  %s\n", status.arr);
if (new_bal >= 0)
    printf("   Balance is now:  $%.2f\n", new_bal);
}  /* End of loop */

Interactive Session

Embedded PL/SQL Debit Transaction Demo

Trying to connect... connected.

** Debit which account number? (-1 to end) 1
  What is the debit amount? 300

  Status:  Transaction completed.
  Balance is now:  $700.00
** Debit which account number? (-1 to end) 1
What is the debit amount? 900
Status: Insufficient funds.
Balance is now: $700.00

** Debit which account number? (-1 to end) 2
What is the debit amount? 500
Status: Transaction completed.
Balance is now: $1500.00

** Debit which account number? (-1 to end) 2
What is the debit amount? 100
Status: Transaction completed.
Balance is now: $1400.00

** Debit which account number? (-1 to end) 99
What is the debit amount? 100
Status: Account not found.

** Debit which account number? (-1 to end) -1

Output Tables

SQL> SELECT * FROM accounts ORDER BY account_id;

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>BAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>700</td>
</tr>
<tr>
<td>2</td>
<td>1400</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>6500</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
</tr>
</tbody>
</table>

SQL> SELECT * FROM journal ORDER BY date_tag;

<table>
<thead>
<tr>
<th>ACCOUNT_ID</th>
<th>ACTION</th>
<th>AMOUNT</th>
<th>DATE_TAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Debit</td>
<td>300</td>
<td>28-NOV-88</td>
</tr>
<tr>
<td>2</td>
<td>Debit</td>
<td>500</td>
<td>28-NOV-88</td>
</tr>
<tr>
<td>2</td>
<td>Debit</td>
<td>100</td>
<td>28-NOV-88</td>
</tr>
</tbody>
</table>
Sample 6. Calling a Stored Procedure

This Pro*C program connects to Oracle, prompts the user for a department number, then calls procedure get_employees, which is stored in package personnel. The procedure declares three index-by tables as OUT formal parameters, then fetches a batch of employee data into the index-by tables. The matching actual parameters are host arrays.

When the procedure finishes, it automatically assigns all row values in the index-by tables to corresponding elements in the host arrays. The program calls the procedure repeatedly, displaying each batch of employee data, until no more data is found.

Input Table

SQL> SELECT ename, empno, sal FROM emp ORDER BY sal DESC;

<table>
<thead>
<tr>
<th>ENAME</th>
<th>EMPNO</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KING</td>
<td>7839</td>
<td>5000</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7788</td>
<td>3000</td>
</tr>
<tr>
<td>FORD</td>
<td>7902</td>
<td>3000</td>
</tr>
<tr>
<td>JONES</td>
<td>7566</td>
<td>2975</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7698</td>
<td>2850</td>
</tr>
<tr>
<td>CLARK</td>
<td>7782</td>
<td>2450</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7499</td>
<td>1600</td>
</tr>
<tr>
<td>TURNER</td>
<td>7844</td>
<td>1500</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
<td>1300</td>
</tr>
<tr>
<td>WARD</td>
<td>7521</td>
<td>1250</td>
</tr>
<tr>
<td>MARTIN</td>
<td>7654</td>
<td>1250</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>1100</td>
</tr>
<tr>
<td>JAMES</td>
<td>7900</td>
<td>950</td>
</tr>
<tr>
<td>SMITH</td>
<td>7369</td>
<td>800</td>
</tr>
</tbody>
</table>

Stored Procedure

/* available online in file 'sample6' */
#include <stdio.h>
#include <string.h>

typedef char asciz;
EXEC SQL BEGIN DECLARE SECTION;

/* Define type for null-terminated strings. */
EXEC SQL TYPE asciz IS STRING(20);
asciz  username[20];
asciz  password[20];
int    dept_no;    /* which department to query */
char   emp_name[10][21];
char   job[10][21];
EXEC SQL VAR emp_name is STRING (21);
EXEC SQL VAR job is STRING (21);
float  salary[10];
int    done_flag;
int    array_size;
int    num_ret;    /* number of rows returned */
int    SQLCODE;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;

int print_rows();       /* produces program output */
int sqlerror();         /* handles unrecoverable errors */

main()
{
    int i;

    /* Connect to Oracle. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");

    EXEC SQL WHENEVER SQLERROR DO sqlerror();

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("%nConnected to Oracle as user: %s%n", username);

    printf("Enter department number: ");
    scanf("%d", &dept_no);
    fflush(stdin);

    /* Print column headers. */
    printf("%-10.10s%-10.10s%s
" , "Employee", "Job", "Salary");
    printf("%-10.10s%-10.10s%-10.10s%s
" , "--------", "---", "------");
/* Set the array size. */
array_size = 10;

done_flag = 0;

num_ret = 0;

/* Array fetch loop - ends when NOT FOUND becomes true. */
for (;;)
{
    EXEC SQL EXECUTE
    BEGIN personnel.get_employees
        (:dept_no, :array_size, :num_ret, :done_flag,
         :emp_name, :job, :salary);
    END;
    END-EXEC;

    print_rows(num_ret);

    if (done_flag)
        break;
}

/* Disconnect from Oracle. */
EXEC SQL COMMIT WORK RELEASE;
exit(0);
}

print_rows(n)
int n;
{
    int i;

    if (n == 0)
    {
        printf("No rows retrieved.\n");
        return;
    }

    for (i = 0; i < n; i++)
        printf("%10.10s%10.10s%6.2f\n",
               emp_name[i], job[i], salary[i]);
}
Sample 6. Calling a Stored Procedure

```sql
sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\nOracle error detected:"");
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

Interactive Session

Connected to Oracle as user: SCOTT

Enter department number: 20

<table>
<thead>
<tr>
<th>Employee</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>CLERK</td>
<td>800.00</td>
</tr>
<tr>
<td>JONES</td>
<td>MANAGER</td>
<td>2975.00</td>
</tr>
<tr>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000.00</td>
</tr>
<tr>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100.00</td>
</tr>
<tr>
<td>FORD</td>
<td>ANALYST</td>
<td>3000.00</td>
</tr>
</tbody>
</table>
This appendix explains the semantic differences between the \texttt{CHAR} and \texttt{VARCHAR2} base types. These subtle but important differences come into play when you assign, compare, insert, update, select, or fetch character values.

This appendix discusses the following topics:

- Assigning Character Values
- Comparing Character Values
- Inserting Character Values
- Selecting Character Values
Assigning Character Values

When you assign a character value to a CHAR variable, if the value is shorter than the declared length of the variable, PL/SQL blank-pads the value to the declared length. So, information about trailing blanks is lost. In the following example, the value assigned to last_name includes six trailing blanks, not just one:

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

If the character value is longer than the declared length of the CHAR variable, PL/SQL aborts the assignment and raises the predefined exception VALUE_ERROR. PL/SQL neither truncates the value nor tries to trim trailing blanks. For example, given the declaration

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

the following assignment raises VALUE_ERROR:

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

When you assign a character value to a VARCHAR2 variable, if the value is shorter than the declared length of the variable, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, so no information is lost. If the character value is longer than the declared length of the VARCHAR2 variable, PL/SQL aborts the assignment and raises VALUE_ERROR. PL/SQL neither truncates the value nor tries to trim trailing blanks.

Comparing Character Values

You can use the relational operators to compare character values for equality or inequality. Comparisons are based on the collating sequence used for the database character set. One character value is greater than another if it follows it in the collating sequence. For example, given the declarations

```sql
last_name1 VARCHAR2(10) := 'COLES';
last_name2 VARCHAR2(10) := 'COLEMAN';
```

the following IF condition is true:

```sql
IF last_name1 > last_name2 THEN ...
```
ANSI/ISO SQL requires that two character values being compared have equal lengths. So, if both values in a comparison have datatype \texttt{CHAR}, \textit{blank-padding} semantics are used. That is, before comparing character values of unequal length, PL/SQL blank-pads the shorter value to the length of the longer value. For example, given the declarations

```sql
last_name1 CHAR(5) := 'BELLO';
last_name2 CHAR(10) := 'BELLO   ';  -- note trailing blanks
```

the following \texttt{IF} condition is true:

```sql
IF last_name1 = last_name2 THEN ...
```

If either value in a comparison has datatype \texttt{VARCHAR2}, \textit{non-blank-padding} semantics are used. That is, when comparing character values of unequal length, PL/SQL makes no adjustments and uses the exact lengths. For example, given the declarations

```sql
last_name1 VARCHAR2(10) := 'DOW';
last_name2 VARCHAR2(10) := 'DOW   ';  -- note trailing blanks
```

the following \texttt{IF} condition is false:

```sql
IF last_name1 = last_name2 THEN ...
```

If one value in a comparison has datatype \texttt{VARCHAR2} and the other value has datatype \texttt{CHAR}, non-blank-padding semantics are used. But, remember, when you assign a character value to a \texttt{CHAR} variable, if the value is shorter than the declared length of the variable, PL/SQL blank-pads the value to the declared length. So, given the declarations

```sql
last_name1 VARCHAR2(10) := 'STAUB';
lst_name2 CHAR(10) := 'STAUB';  -- PL/SQL blank-pads value
```

the following \texttt{IF} condition is false because the value of \texttt{last_name2} includes five trailing blanks:

```sql
IF last_name1 = last_name2 THEN ...
```

All string literals have datatype \texttt{CHAR}. So, if both values in a comparison are literals, blank-padding semantics are used. If one value is a literal, blank-padding semantics are used only if the other value has datatype \texttt{CHAR}. 
Inserting Character Values

When you insert the value of a PL/SQL character variable into an Oracle database column, whether the value is blank-padded or not depends on the column type, not on the variable type.

When you insert a character value into a CHAR database column, Oracle does not strip trailing blanks. If the value is shorter than the defined width of the column, Oracle blank-pads the value to the defined width. As a result, information about trailing blanks is lost. If the character value is longer than the defined width of the column, Oracle aborts the insert and generates an error.

When you insert a character value into a VARCHAR2 database column, Oracle does not strip trailing blanks. If the value is shorter than the defined width of the column, Oracle does not blank-pad the value. Character values are stored intact, so no information is lost. If the character value is longer than the defined width of the column, Oracle aborts the insert and generates an error.

Note: The same rules apply when updating.

When inserting character values, to ensure that no trailing blanks are stored, use the function RTRIM, which trims trailing blanks. An example follows:

```sql
DECLARE
    my_name VARCHAR2(15);
BEGIN
    my_ename := 'LEE   ';  -- note trailing blanks
    INSERT INTO emp
        VALUES (my_empno, RTRIM(my_ename), ...);  -- inserts 'LEE'
END;
```

Selecting Character Values

When you select a value from an Oracle database column into a PL/SQL character variable, whether the value is blank-padded or not depends on the variable type, not on the column type.

When you select a column value into a CHAR variable, if the value is shorter than the declared length of the variable, PL/SQL blank-pads the value to the declared length. As a result, information about trailing blanks is lost. If the character value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR.
When you select a column value into a VARCHAR2 variable, if the value is shorter than the declared length of the variable, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are stored intact, so no information is lost.

For example, when you select a blank-padded CHAR column value into a VARCHAR2 variable, the trailing blanks are not stripped. If the character value is longer than the declared length of the VARCHAR2 variable, PL/SQL aborts the assignment and raises VALUE_ERROR.

**Note:** The same rules apply when fetching.
This appendix shows you how to run the Wrap Utility, a standalone programming utility that encrypts PL/SQL source code. You can use the Wrap Utility to deliver PL/SQL applications without exposing your source code.

This appendix discusses the following topics:

- Advantages of Wrapping PL/SQL Procedures
- Running the Wrap Utility
- Guidelines

**Advantages of Wrapping PL/SQL Procedures**

By hiding application internals, the Wrap Utility prevents

- Misuse of your application by other developers
- Exposure of your algorithms to business competitors

Wrapped code is as portable as source code. The PL/SQL compiler recognizes and loads wrapped compilation units automatically. Other advantages include

- Platform independence—you need not deliver multiple versions of the same compilation unit
- Dynamic loading—users need not shut down and re-link to add a new feature
- Dynamic binding—external references are resolved at load time
- Strict dependency checking—invalidated program units are recompiled automatically
Running the Wrap Utility

To run the Wrap Utility, enter the `wrap` command at your operating system prompt using the following syntax:

```
wrap iname=input_file [oname=output_file]
```

Leave no space around the equal signs because spaces delimit individual arguments.

The `wrap` command requires only one argument, which is `iname=input_file`

where `input_file` is the name of the Wrap Utility input file. You need not specify the file extension because it defaults to `sql`. For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql
```

However, you can specify a different file extension as the following example shows:

```
wrap iname=/mydir/myfile.src
```

Optionally, the `wrap` command takes a second argument, which is `oname=output_file`

where `output_file` is the name of the Wrap Utility output file. You need not specify the output file because its name defaults to that of the input file and its extension defaults to `plb` (PL/SQL binary). For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

However, you can use the option `oname` to specify a different file name and extension, as the following example shows:

```
wrap iname=/mydir/myfile oname=/yourdir/yourfile.obj
```
Input and Output Files for the Wrap Utility

The input file can contain any combination of SQL statements. However, the Wrap Utility encrypts only the following CREATE statements, which define subprograms, packages, or object types:

- `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 ... OBJECT`
- `CREATE [OR REPLACE] TYPE BODY type_name`

All other SQL statements are passed intact to the output file. Comment lines are deleted unless they appear inside a subprogram, package, or object type.

When encrypted, a subprogram, package, or object type has the form

```plaintext
<header> wrapped <body>
```

where `header` begins with the reserved word `CREATE` and ends with the name of the subprogram, package, or object type, and `body` is an intermediate form of object code. The word `wrapped` tells the PL/SQL compiler that the subprogram, package, or object type was encrypted by the Wrap Utility.

The header can contain comments. For example, the Wrap Utility converts

```plaintext
CREATE PACKAGE
-- Author: J. Hollings
-- Date:   10/15/99
banking AS
    minimum_balance CONSTANT REAL := 25.00;
    insufficient_funds EXCEPTION;
END banking;
```

into

```plaintext
CREATE PACKAGE
-- Author: J. Hollings
-- Date:   10/15/99
banking wrapped
0
abcd ...
```

Generally, the output file is much larger than the input file.
Guidelines

Error Handling in the Wrap Utility

If your input file contains syntax errors, the Wrap Utility detects and reports them. However, the Wrap Utility cannot detect semantic errors because it does not resolve external references. For example, it does not report the following error (table or view does not exist):

```sql
CREATE PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER) AS
BEGIN
    UPDATE amp  -- should be emp
    SET sal = sal + amount WHERE empno = emp_id;
END;
```

The PL/SQL compiler resolves external references. So, semantic errors are reported when the Wrap Utility output file (.plb file) is compiled.

Version Compatibility

The Wrap Utility is upward-compatible with Oracle. So, for example, you can load files processed by the V8.1.5 Wrap Utility into a V8.1.6 Oracle database. However, the Wrap Utility is not downward-compatible with Oracle. So, for example, you cannot load files processed by the V8.1.6 Wrap Utility into a V8.1.5 Oracle database.

Guidelines

When wrapping a package or object type, wrap only the body, not the spec. That way, other developers see the information they need to use the package or type, but they do not see its implementation.

Like all encrypted files, wrapped files cannot be edited. To revise a wrapped file, you must revise and re-wrap the underlying source code. So, do not wrap a subprogram, package, or object type until it is ready for shipment to end-users.
This appendix explains how PL/SQL resolves references to names in potentially ambiguous SQL and procedural statements.

This appendix discusses the following topics:

What Is Name Resolution?
Various Forms of References
Name-Resolution Algorithm
Understanding Capture
Avoiding Capture
Accessing Attributes and Methods
Calling Subprograms and Methods
Name Resolution for SQL Versus PL/SQL
What Is Name Resolution?

During compilation, the PL/SQL compiler associates identifiers such as the name of a variable with an address (memory location), datatype, or actual value. This process is called *binding*. The association lasts through all subsequent executions until a recompilation occurs, which might cause a rebinding.

Before binding the names, PL/SQL must resolve all references to them in the compilation unit. This process is called *name resolution*. PL/SQL considers all names to be in the same namespace. So, one declaration or definition in an inner scope can hide another in an outer scope. In the following example, the declaration of variable `client` hides the definition of datatype `Client` because PL/SQL is not case sensitive except within string literals:

```sql
BEGIN
  <<block1>>
  DECLARE
      TYPE Client IS RECORD (...);
      TYPE Customer IS RECORD (...);
  BEGIN
      DECLARE
          client Customer; -- hides definition of type Client
          lead1  Client;   -- illegal; Client resolves to the
                           -- variable client
          lead2  block1.Client; -- OK; refers to type Client
      BEGIN
          NULL;
      END;
  END;
END;
```

However, you can still refer to datatype `Client` by qualifying the reference with block label `block1`.

In the `CREATE TYPE person1` statement below, the compiler resolves the second reference to `manager` as the name of the attribute you are trying to declare. In the `CREATE TYPE person2` statement, the compiler resolves the second reference to `manager` as the name of the attribute you just declared. In both cases, the reference to `manager` generates an error because the compiler expects a type name.

```sql
CREATE TYPE manager AS OBJECT (dept NUMBER);
CREATE TYPE person1 AS OBJECT (manager manager);
CREATE TYPE person2 AS OBJECT (manager NUMBER, mgr manager);
```
Various Forms of References

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. Some examples of legal references follow:

```
CREATE PACKAGE pkg1 AS
  m NUMBER;
  TYPE t1 IS RECORD (a NUMBER);
  v1 t1;
  TYPE t2 IS TABLE OF t1 INDEX BY BINARY_INTEGER;
  v2 t2;
  FUNCTION f1 (p1 NUMBER) RETURN t1;
  FUNCTION f2 (q1 NUMBER) RETURN t2;
END pkg1;

CREATE PACKAGE BODY pkg1 AS
  FUNCTION f1 (p1 NUMBER) RETURN t1 IS
    n NUMBER;
    BEGIN
      n := m;            -- (1) unqualified name
      n := pkg1.m;       -- (2) dot-separated chain of identifiers
                         -- (package name used as scope
                         -- qualifier followed by variable name)
      n := pkg1.f1.p1;   -- (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 := v1.a;         -- (4) dot-separated chain of identifiers
                         -- (variable name followed by
                         -- component selector)
      n := pkg1.v1.a;    -- (5) dot-separated chain of identifiers
                         -- (package name used as scope
                         -- qualifier followed by
                         -- variable name followed by component
                         -- selector)
      n := v2(10).a;     -- (6) indexed name followed by component
                         -- selector
      n := f1(10).a;     -- (7) function call followed by component
                         -- selector
      n := f2(10)(10).a; -- (8) function call followed by indexing
                         -- followed by component selector
      n := scott.pkg1.f2(10)(10).a;
```
Name-Resolution Algorithm

Let us take a look at the name-resolution algorithm.

The first part of name resolution involves finding the basis. The basis is the smallest prefix to a dot-separated chain of identifiers that can be resolved by looking in the current scope, then moving outward to schema-level scopes.

In the previous examples, the basis for (3) pkg1.f1.p1 is pkg1, the basis for (4) scott.pkg1.f1.n is scott.pkg1, and the basis for (5) v1.a is v1. In (5), the a in v1.a is a component selector and resolves as field a of variable v1 because v1 is of type t1, which has a field called a.

If a basis is not found, the compiler generates a not declared error. If the basis is found, the compiler attempts to resolve the complete reference. If it fails, the compiler generates an error.
The length of the basis is always 1, 2, or 3. And, it can be 3 only inside SQL scope when the compiler resolves a three-part name as

```
 schema_name.table_name.column_name
```

Here are more examples of bases:

```
variable_name
type_name
package_name
schema_name.package_name
schema_name.function_name
table_name
table_name.column_name
schema_name.table_name
schema_name.table_name.column_name
```

### Finding the Basis

Now, let us look at the algorithm for finding the basis.

If the compiler is resolving a name in SQL scope (which includes everything in a DML statement except items in the \texttt{INTO} clause and schema-level table names), it first attempts to find the basis in that scope. If it fails, it attempts to find the basis in PL/SQL local scopes and at the schema level just as it would for names in non-SQL scopes.

Here are the rules for finding the basis in SQL scope when the compiler expects to find a column name:

- Given one identifier, the compiler attempts to find a basis of length 1 using the identifier as an unqualified column name in one of the tables listed in any \texttt{FROM} clauses that are in scope, starting with the current scope and moving outward.

- Given a chain of two identifiers, the compiler attempts to find a basis of length 2 using the identifiers as a column name qualified by a table name or table alias, starting with the current scope and moving outward.

- Given a chain of three identifiers, the compiler attempts to find in each scope that it searches, starting with the current scope and moving outward, either
  - a basis of length 3 using the three identifiers as a column name qualified by a table name qualified by a schema name, or
  - a basis of length 2 using the first two identifiers as a column name of some user-defined type qualified by a table alias
Given a chain of four identifiers, the compiler attempts to find a basis of length 2, using the first two identifiers as a column name of some user-defined type qualified by a table alias, starting with the current scope and moving outward.

Once the compiler finds the basis as a column name, it attempts to resolve the complete reference by finding a component of the basis and so on depending upon the type of column name.

Here are the rules for finding the basis in SQL scope when the compiler expects to find a row expression (which is a table alias that can appear by itself; it can be used only with an object table and operator `REF` or `VALUE`, or in an `INSERT` or `UPDATE` statement for an object table):

- Given one identifier, the compiler attempts to find a basis of length 1 as a table alias, starting with the current scope and moving outward. If the table alias does not correspond to an object table, the compiler generates an error.
- Given a chain of two or more identifiers, the compiler generates an error.
- If the name being resolved either does not appear in SQL scope, or appears in SQL scope but the compiler cannot find a basis for it in that scope, the compiler attempts to find the basis by searching all PL/SQL scopes local to the compilation unit, starting with the current scope and moving outward. If the name is found, the length of the basis is 1. If the name is not found, the compiler attempts to find the basis by searching for schema objects using the following rules:
  1. First, the compiler attempts to find a basis of length 1 by searching the current schema for a schema object whose name matches the first identifier in the chain of identifiers. The schema object found might be a package specification, function, procedure, table, view, sequence, synonym, or schema-level datatype. If it is a synonym, the basis will be resolved as the base object designated by the synonym.
  2. If the previous search fails, the compiler attempts to find a basis of length 1 by searching for a public synonym whose name matches the first identifier in the chain. If this succeeds, the basis will be resolved as the base object designated by the synonym.
  3. If the previous search fails and there are at least two identifiers in the chain, the compiler attempts to find a basis of length 2 by searching for a schema object whose name matches the second identifier in the chain and which is owned by a schema whose name matches the first identifier in the chain.
4. If the compiler finds a basis as a schema object, it checks the privileges on the base object. If the base object is not visible, the compiler generates a *not declared* error because an *insufficient privileges* error would acknowledge the existence of the object, which is a security violation.

5. If the compiler fails to find a basis by searching for schema objects, it generates a *not declared* error.

6. If the compiler finds a basis, it attempts to resolve the complete reference depending on how the basis was resolved. If it fails to resolve the complete reference, the compiler generates an error.

Understanding Capture

When a declaration or type definition in another scope prevents the compiler from resolving a reference correctly, that declaration or definition is said to "capture" the reference. Usually this is the result of migration or schema evolution. There are three kinds of capture: inner, same-scope, and outer. Inner and same-scope capture apply only in SQL scope.

Inner Capture

An inner capture occurs when a name in an inner scope that once resolved to an entity in an outer scope, either

- gets resolved to an entity in an inner scope, or
- causes an error because the basis of the identifier chain got captured in an inner scope and the complete reference could not be resolved

If the situation was resolved without error in an inner scope, the capture might occur without your knowing. 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);
CREATE TABLE tab2 (col1 NUMBER);
CREATE PROCEDURE proc AS
    CURSOR c1 IS SELECT * FROM tab1
        WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
    ...
END;
```
In the last example, if you add a column named \texttt{col2} to table \texttt{tab2}, as follows

\begin{verbatim}
ALTER TABLE tab2 ADD (col2 NUMBER);
\end{verbatim}

then procedure \texttt{proc} is invalidated and recompiled automatically upon next use. However, upon recompilation, the \texttt{col2} in the inner \texttt{SELECT} statement binds to column \texttt{col2} in table \texttt{tab2} because \texttt{tab2} is in the inner scope. Thus, the reference to \texttt{col2} is captured by the addition of column \texttt{col2} to table \texttt{tab2}.

The use of collections and object types allows for more inner capture situations. In the following example, the reference to \texttt{s.tab2.a} resolves to attribute \texttt{a} of column \texttt{tab2} in table \texttt{tab1} through table alias \texttt{s}, which is visible in the outer scope of the query:

\begin{verbatim}
CREATE TYPE type1 AS OBJECT (a NUMBER);
CREATE TABLE tab1 (tab2 type1);
CREATE TABLE tab2 (x NUMBER);
SELECT * FROM tab1 s  -- alias with same name as schema name
    WHERE EXISTS (SELECT * FROM s.tab2 WHERE x = s.tab2.a);
    -- note lack of alias
\end{verbatim}

In the last example, suppose you add a column named \texttt{a} to table \texttt{s.tab2}, which appears in the inner subquery. When the query is processed, an inner capture occurs because the reference to \texttt{s.tab2.a} resolves to column \texttt{a} of table \texttt{tab2} in schema \texttt{s}. You can avoid inner captures by following the rules given in "Avoiding Capture" on page D-9. According to those rules, you should revise the query as follows:

\begin{verbatim}
SELECT * FROM s.tab1 p1
    WHERE EXISTS (SELECT * FROM s.tab2 p2 WHERE p2.x = p1.tab2.a);
\end{verbatim}

### Same-Scope Capture

In SQL scope, a same-scope capture occurs when a column is added to one of two tables in the same scope, and that column has the same name as a column in the other table. Consider the following query (and refer to the previous example):

\begin{verbatim}
PROCEDURE proc IS
    CURSOR c1 IS SELECT * FROM tab1, tab2 WHERE col2 = 10;
\end{verbatim}

In this example, the reference to \texttt{col2} in the query binds to column \texttt{col2} in table \texttt{tab1}. If you add a column named \texttt{col2} to table \texttt{tab2}, the query compiles with errors. Thus, the reference to \texttt{col2} is captured by an error.
Outer Capture

An outer capture occurs when a name in an inner scope, which once resolved to an entity in an inner scope, gets resolved to an entity in an outer scope. Fortunately, SQL and PL/SQL are designed to prevent outer captures.

Avoiding Capture

You can avoid inner capture in DML statements by following these rules:

- Specify an alias for each table in the DML statement.
- Keep table aliases unique throughout the DML statement.
- Avoid table aliases that match schema names used in the query.
- Qualify each column reference with the table alias.

Qualifying a reference with `<schema-name>.<table-name>` does not prevent inner capture if the DML statement references tables that have columns of a user-defined object type.

Accessing Attributes and Methods

Columns of a user-defined object type allow for more inner capture situations. To minimize problems, the name-resolution algorithm includes the following rules:

- All references to attributes and methods must be qualified by a table alias. So, 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 illegal if they are prefixed with a table name:

```sql
CREATE TYPE t1 AS OBJECT (x NUMBER);
CREATE TABLE tbl (col t1);
SELECT col.x FROM tbl; -- illegal
SELECT tbl.col.x FROM tbl; -- illegal
SELECT scott.tbl.col.x FROM scott.tbl; -- illegal
SELECT t.col.x FROM tbl t;
UPDATE tbl SET col.x = 10; -- illegal
UPDATE scott.tbl SET scott.tbl.col.x=10; -- illegal
UPDATE tbl t SET t.col.x = 1;
DELETE FROM tbl WHERE tbl.col.x = 10; -- illegal
DELETE FROM tbl t WHERE t.col.x = 10;
```
Row expressions must resolve as references to table aliases. You can pass row expressions to operators REF and VALUE, and you can use row expressions in the SET clause of an UPDATE statement. Some examples follow:

```
CREATE TYPE t1 AS OBJECT (x number);
CREATE TABLE ot1 OF t1;                       -- object table
SELECT REF(ot1) FROM ot1;                     -- illegal
SELECT REF(o) FROM ot1 o;
SELECT VALUE(ot1) FROM ot1;                   -- illegal
SELECT VALUE(o) FROM ot1 o;
DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10));  -- illegal
DELETE FROM ot1 o WHERE VALUE(o) = (t1(10));
UPDATE ot1 SET ot1 = ...                      -- illegal
UPDATE ot1 o SET o = ....
```

The following ways to insert into an object table are legal and do not require an alias because there is no column list:

```
INSERT INTO ot1 VALUES (t1(10)); -- no row expression
INSERT INTO ot1 VALUES (10);     -- no row expression
```

Calling Subprograms and Methods

In calls to a parameterless subprogram, an empty parameter list is optional, as the following examples show:

```
CREATE FUNCTION func1 RETURN NUMBER AS
BEGIN
   RETURN 10;
END;

CREATE PACKAGE pkg2 AS
   FUNCTION func1 RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(func1,WNDS,RNDS,WNPS,RNPS);
END pkg2;

CREATE PACKAGE BODY pkg2 AS
   FUNCTION func1 RETURN NUMBER IS
      BEGIN
         RETURN 20;
      END;
END pkg2;
```
SELECT func1 FROM dual;
SELECT func1() FROM dual;

SELECT pkg2.func1 FROM dual;
SELECT pkg2.func1() FROM dual;

DECLARE
  x NUMBER;
BEGIN
  x := func1;
  x := func1();
  SELECT func1 INTO x FROM dual;
  SELECT func1() INTO x FROM dual;
  SELECT pkg2.func1 INTO x FROM dual;
  SELECT pkg2.func1() INTO x FROM dual;
END;

In calls to a parameterless method, an empty parameter list is optional within PL/SQL scopes. However, an empty parameter list is required within SQL scopes. Some examples follow:

CREATE TYPE type1 AS OBJECT (  
a NUMBER,
  MEMBER FUNCTION f RETURN number,
  PRAGMA RESTRICT_REFERENCES(f,WNDS,RNDS,WNPS,RNPS)
);

CREATE TYPE BODY type1 AS
  MEMBER FUNCTION f RETURN number IS
    BEGIN
      RETURN 1;
    END;
END;

CREATE TABLE tab1 (col1 type1);
INSERT INTO tab1 VALUES (type1(10));
SELECT x.col1.f FROM tab1 x;   -- illegal
SELECT x.col1.f() FROM tab1 x;
Name Resolution for SQL Versus PL/SQL

The name-resolution rules for SQL and PL/SQL are similar. However, there are a few minor differences, which are not noticeable if you follow the capture avoidance rules.

For compatibility, the SQL rules are more permissive than the PL/SQL rules. That is, the SQL rules, which are mostly context sensitive, recognize as legal more situations and DML statements than the PL/SQL rules do.
PL/SQL is designed primarily for high-speed transaction processing. That design imposes some program limits, which are discussed in this appendix.

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), which is a tree-structured intermediate language. It is defined using a meta-notation called Interface Definition Language (IDL). DIANA provides for communication internal to compilers and other tools.

At compile time, PL/SQL source code is translated into machine-readable m-code. Both the DIANA and m-code for a procedure 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 procedures; the m-code is simply executed.

In the shared memory pool, a package spec, object type spec, standalone subprogram, or anonymous block is limited to $2^{26}$ 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 E–1.

**Table E–1  PL/SQL Compiler Limits**

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind variables passed to a program unit</td>
<td>32K</td>
</tr>
<tr>
<td>exception handlers in a program unit</td>
<td>64K</td>
</tr>
<tr>
<td>fields in a record</td>
<td>64K</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>magnitude of a <strong>BINARY_INTEGER</strong> value</td>
<td>2G</td>
</tr>
</tbody>
</table>
To estimate how much memory a program unit requires, you can query the data dictionary view `user_object_size`. The column `parsed_size` returns the size (in bytes) of the "flattened" DIANA. In the following example, you get the parsed size of a package (displayed on the package spec line):

```plsql
CREATE PACKAGE pkg1 AS
    PROCEDURE proc1;
END pkg1;
/

CREATE PACKAGE BODY pkg1 AS
    PROCEDURE proc1 IS
    BEGIN
        NULL;
    END;
END pkg1;
```

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>magnitude of a PLS_INTEGER value</td>
<td>2G</td>
</tr>
<tr>
<td>objects referenced by a program unit</td>
<td>64K</td>
</tr>
<tr>
<td>parameters passed to an explicit cursor</td>
<td>64K</td>
</tr>
<tr>
<td>parameters passed to a function or procedure</td>
<td>64K</td>
</tr>
<tr>
<td>precision of a FLOAT value (binary digits)</td>
<td>126</td>
</tr>
<tr>
<td>precision of a REAL value (binary digits)</td>
<td>63</td>
</tr>
<tr>
<td>size of an identifier (characters)</td>
<td>30</td>
</tr>
<tr>
<td>size of a string literal (bytes)</td>
<td>32K</td>
</tr>
<tr>
<td>size of a CHAR value (bytes)</td>
<td>32K</td>
</tr>
<tr>
<td>size of a LONG value (bytes)</td>
<td>32K-7</td>
</tr>
<tr>
<td>size of a LONG RAW value (bytes)</td>
<td>32K-7</td>
</tr>
<tr>
<td>size of a RAN value (bytes)</td>
<td>32K</td>
</tr>
<tr>
<td>size of a VARCHAR2 value (bytes)</td>
<td>32K</td>
</tr>
<tr>
<td>size of an NCHAR value (bytes)</td>
<td>32K</td>
</tr>
<tr>
<td>size of an NVARCHAR2 value (bytes)</td>
<td>32K</td>
</tr>
<tr>
<td>size of a BFILE value (bytes)</td>
<td>4G</td>
</tr>
<tr>
<td>size of a BLOB value (bytes)</td>
<td>4G</td>
</tr>
<tr>
<td>size of a CLOB value (bytes)</td>
<td>4G</td>
</tr>
<tr>
<td>size of an NCLOB value (bytes)</td>
<td>4G</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 *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.
The words listed in this appendix are reserved by PL/SQL. That is, they have a special syntactic meaning to PL/SQL. So, you should not use them to name program objects such as constants, variables, or cursors. Some of these words (marked by an asterisk) are also reserved by SQL. So, you should not use them to name schema objects such as columns, tables, or indexes.
Symbols

%BULK_EXCEPTIONS cursor attribute, 5-38
%BULK_ROW_COUNT cursor attribute, 5-36
%FOUND cursor attribute, 6-36, 6-39
%ISOPEN cursor attribute, 6-35, 6-39
%NOTFOUND cursor attribute, 6-35
%ROW_COUNT cursor attribute, 6-36, 6-39
+ addition/identity operator, 2-3
:= assignment operator, 1-4, 2-4
=> association operator, 2-4, 8-13
% attribute indicator, 1-7, 2-3
' character string delimiter, 2-3
. component selector, 1-6, 2-3
| | concatenation operator, 2-4, 2-28
/ division operator, 2-3
** exponentiation operator, 2-4
(expression or list delimiter, 2-3
) expression or list delimiter, 2-3
: host variable indicator, 2-3
.item separator, 2-3
<< label delimiter, 2-4
>> label delimiter, 2-4
/* multi-line comment delimiter, 2-4
*/ multi-line comment delimiter, 2-4
* multiplication operator, 2-3
" quoted identifier delimiter, 2-3
.. range operator, 2-4, 4-13
= relational operator, 2-3, 2-27
< relational operator, 2-3, 2-27
> relational operator, 2-3, 2-27
<> relational operator, 2-4, 2-27
!= relational operator, 2-4, 2-27
^ relational operator, 2-4
<= relational operator, 2-4, 2-27
>= relational operator, 2-4, 2-27
@ remote access indicator, 2-3, 2-17
-- single-line comment delimiter, 2-4
; statement terminator, 2-3, 13-16
- subtraction/negation operator, 2-3

A

abstraction, 8-3, 10-2
ACCESS_INTO_NULL exception, 7-5
actual parameters, 6-8
address, 6-16
aggregate assignment, 2-15
aggregate functions
  AVG, 6-2
  COUNT, 6-2
  GROUPING, 6-2
  MAX, 6-2
  MIN, 6-2
  STDDEV, 6-2
  SUM, 6-2
  treatment of nulls, 6-2
  VARIANCE, 6-2

AL16UTF16 character encoding, 3-11
aliasing, 8-21
ALL row operator, 6-3, 6-5
ALTER TYPE statement
  for type evolution, 10-12
anonymous PL/SQL block, 8-2
ANY comparison operator, 6-5
apostrophes, 2-9
architecture, 1-17
assignment operator, 1-4
assignment statement
   syntax, 13-4
assignments
   aggregate, 2-15
   character string, B-2
   collection, 5-11
   cursor variable, 6-31
   field, 5-51
   record, 5-51
   semantics, B-2
association operator, 8-13
asterisk (*) row operator, 6-3
asynchronous operations, 9-16
atomically null, 10-25
attribute indicator, 1-7
attributes, 1-7
   %ROWTYPE, 2-14
   %TYPE, 2-13
   cursor, 6-34
   object, 10-3, 10-7
AUTHID clause, 8-4, 8-6, 8-50
autonomous transactions, 6-53
   advantages, 6-53
   controlling, 6-57
autonomous triggers, 6-59
AUTONOMOUS_TRANSACTION pragma, 6-54
   syntax, 13-8
AVG aggregate function, 6-2

B
base types, 3-3, 3-21
basic loops, 4-9
BETWEEN comparison operator, 2-27, 6-5
BFILE datatype, 3-14
binary operators, 2-23
BINARY_INTEGER datatype, 3-3
bind argument, 11-3
binding, 5-33
blank-padding semantics, B-3
BLOB datatype, 3-14
blocks
   anonymous, 8-2
   label, 2-21
PL/SQL, 13-11
   structure, 1-2
body
   cursor, 6-12
   function, 8-7
   method, 10-8
   object, 10-5
   package, 9-8
   procedure, 8-4
BOOLEAN datatype, 3-15
Boolean expressions, 2-28
Boolean literals, 2-9
built-in functions, 2-35
bulk binds, 5-32
BULK COLLECT clause, 5-41
bulk fetches, 5-42
bulk returns, 5-43
%BULK_EXCEPTIONS cursor attribute, 5-38
%BULK_ROWCOUNT cursor attribute, 5-36
by-reference parameter passing, 8-21
by-value parameter passing, 8-21

C
call specification, 9-3
calls
   inter-language, 8-63
   subprogram, 8-13
carriage returns, 2-2
CASE expressions, 2-31
case sensitivity
   identifier, 2-5
   string literal, 2-9
CASE statement, 4-5
   syntax, 13-18
CASE_NOT_FOUND exception, 7-5
CHAR
   datatype, 3-5
   maximum width of column, 3-5
   semantics, B-1
character literals, 2-8
character sets, 2-2
CHARACTER subtype, 3-6
character values
   assigning, B-2
comparing, B-2
inserting, B-4
selecting, B-4

clauses
  AUTHID, 8-4, 8-6, 8-50
  BULK COLLECT, 5-41
  LIMIT, 5-43

client programs, 10-2
CLOB datatype, 3-14
CLOSE statement, 6-11, 6-24
  syntax, 13-21
collating sequence, 2-29
collection exceptions
  when raised, 5-30
collection methods
  applying to parameters, 5-29
  COUNT, 5-24
  DELETE, 5-28, 13-24
  EXISTS, 5-24
  EXTEND, 5-27
  FIRST, 5-25
  LAST, 5-25
  LIMIT, 5-25
  NEXT, 5-26
  PRIOR, 5-26
  syntax, 13-23
  TRIM, 5-27

COLLECTION_IS_NULL exception, 7-5
collections, 5-2
  assigning, 5-11
  bulk binding, 5-32
  comparing, 5-13
  constructors, 5-8
  declaring, 5-7
  defining, 5-5
  element types, 5-5
  initializing, 5-8
  kinds, 5-1
  multilevel, 5-21
  referencing, 5-10
  scope, 5-8
  syntax, 13-28
column alias, 6-15
  when needed, 2-16

COMMENT clause, 6-45

comments, 2-10
  restrictions, 2-11
  syntax, 13-35

COMMIT statement, 6-44
  syntax, 13-36

comparison operators, 2-26, 6-5

comparisons
  of character values, B-2
  of collections, 5-13
  of expressions, 2-28

compiler hint, NOCOPY, 8-17

compiling PL/SQL procedures for native execution, 12-14

component selector, 1-6

composite types, 3-2

compound symbols, 2-4

concatenation operator, 2-28

treatment of nulls, 2-34

concurrency, 6-43

conditional control, 4-2

constants
  declaring, 2-12
  syntax, 13-38

constraints
  NOT NULL, 2-13
  where not allowed, 8-4

constructors
  collection, 5-8
  object, 10-12

context
  switching, 5-32
  transaction, 6-56

control structures, 4-2
  conditional, 4-2
  iterative, 4-9
  sequential, 4-17

conventions
  naming, 2-17

conversion
  functions, 3-25

conversion, datatype, 3-24

correlated subquery, 6-11

COUNT aggregate function, 6-2

COUNT collection method, 5-24

CURRENT OF clause, 6-50
current row, 1-5
CURRVAL pseudocolumn, 6-3
cursor attributes
  %BULK_EXCEPTIONS, 5-38
  %BULK_ROWCOUNT, 5-36
  %FOUND, 6-35, 6-39
  %ISOPEN, 6-35, 6-39
  %NOTFOUND, 6-35
  %ROWCOUNT, 6-36, 6-39
  implicit, 6-38
  syntax, 13-42
  values, 6-36
cursor expressions, 6-41
cursor FOR loops, 6-13
  passing parameters to, 6-15
cursor subqueries, 6-41
cursor variables, 6-16
  as parameters to table functions, 8-38
  assignment, 6-31
  closing, 6-24
  declaring, 6-18
  fetching from, 6-23
  opening, 6-19
  restrictions, 6-33
  syntax, 13-47
  using to reduce network traffic, 6-30
  using with dynamic SQL, 11-8
CURSOR_ALREADY_OPEN exception, 7-5
cursors, 1-5, 6-6
  analogy, 1-5
  closing, 6-11
  declaring, 6-6
  explicit, 6-6
  fetching from, 6-9
  implicit, 6-11
  opening, 6-7
  packaged, 6-12
  parameterized, 6-8
  RETURN clause, 6-12
  scope rules, 6-7
  syntax, 13-53
dangling refs, 10-35
data
  abstraction, 10-2
  encapsulation, 1-16
  integrity, 6-43
  locks, 6-43
data streaming
  definition, 8-45
database changes
  making permanent, 6-44
  undoing, 6-45
database character set, 3-11
database triggers, 1-19
  autonomous, 6-59
datatypes, 3-1
  BFILE, 3-14
  BINARY_INTEGER, 3-3
  BLOB, 3-14
  BOOLEAN, 3-15
  CHAR, 3-5
  CLOB, 3-14
  constrained, 8-4
  DATE, 3-16
  families, 3-2
  implicit conversion, 3-24
  INTERVAL DAY TO SECOND, 3-20
  INTERVAL YEAR TO MONTH, 3-19
  LONG, 3-6
  LONG RAW, 3-6
  national character, 3-10
  NCHAR, 3-11
  NCLOB, 3-15
  NUMBER, 3-3
  NVARCHAR2, 3-12
  PLS_INTEGER, 3-5
  RAW, 3-7
  RECORD, 5-46
  REF CURSOR, 6-16
  ROWID, 3-7
  scalar versus composite, 3-1
  TABLE, 5-2
  TIMESTAMP, 3-17
  TIMESTAMP WITH LOCAL TIME ZONE, 3-18
  TIMESTAMP WITH TIME ZONE, 3-17
  UROWID, 3-7
  VARCHAR2, 3-9
VARARY, 5-4
DATE datatype, 3-16
dates
  converting, 3-26
  TO_CHAR default format, 3-26
datetime literals, 2-9
DBMS_ALERT package, 9-16
DBMS_OUTPUT package, 9-16
DBMS_PIPE package, 9-17
deadlocks, 6-43
effect on transactions, 6-46
  how broken, 6-46
DEC subtype, 3-4
DECIMAL subtype, 3-4
declarations
  collection, 5-7
  constant, 2-12
  cursor, 6-6
  cursor variable, 6-18
  exception, 7-7
  forward, 8-10
  object, 10-25
  record, 5-48
  subprogram, 8-10
  variable, 2-11
declarative part
  of function, 8-7
  of PL/SQL block, 1-3
  of procedure, 8-4
DECODE function
  treatment of nulls, 2-34
DEFAULT keyword, 2-12
default parameter values, 8-19
define variable, 11-3
definer rights, 8-47
  versus invoker rights, 8-47
DELETE collection method, 5-28, 13-24
DELETE statement
  RETURNING clause, 12-12
  syntax, 13-57
delimiters, 2-3
dense collections, 5-3
DEPT database table, xxviii
DEREF function, 10-36
dereference, 10-36
DETERMINISTIC hint, 8-7
digits of precision, 3-4
DISTINCT operator, 6-3
DISTINCT row operator, 6-3, 6-6
distributed transactions, 6-44
dot notation, 1-6, 1-7
  for collection methods, 5-24
  for global variables, 4-16
  for object attributes, 10-27
  for object methods, 10-28
  for package contents, 9-7
  for record fields, 2-15
DOUBLE PRECISION subtype, 3-4
DUP_VAL_ON_INDEX exception, 7-5
dynamic FOR-loop range, 4-15
dynamic SQL, 11-2
tips and traps, 11-15
  using EXECUTE IMMEDIATE statement, 11-3
  using OPEN-FOR-USING statement, 11-8
dynamic string, 11-3

E
elaboration, 2-12
element types, collection, 5-5
ELSE clause, 4-3
ELSIF clause, 4-4
EMP database table, xxviii
capsulation, data, 1-16
END IF reserved words, 4-3
END LOOP reserved words, 4-11
entended rowids, 3-8
error messages
  maximum length, 7-18
evaluation, 2-23
  short-circuit, 2-26
EXAMPBLD script, A-3
EXAMLPLOD script, A-3
exception handlers, 7-15
  branching from, 7-17
  OTHERS handler, 7-2
  using RAISE statement in, 7-15
  using SQLCODE function in, 7-18
  using SQLERRM function in, 7-18
EXCEPTION_INIT pragma, 7-8
syntax, 13-61
using with raise_application_error, 7-10
exception-handling part
of function, 8-7
of PL/SQL block, 1-3
of procedure, 8-4
exceptions, 7-2
declaring, 7-7
predefined, 7-4
propagation, 7-12
raised in declaration, 7-16
raised in handler, 7-17
raising with RAISE statement, 7-11
reraising, 7-14
scope rules, 7-7
syntax, 13-63
user-defined, 7-7
WHEN clause, 7-15
executable part
of function, 8-7
of PL/SQL block, 1-3
of procedure, 8-4
EXECUTE IMMEDIATE statement, 11-3
EXECUTE privilege, 8-53
execution environments, 1-17
EXISTS collection method, 5-24
EXISTS comparison operator, 6-5
EXIT statement, 4-9, 4-16
syntax, 13-70
WHEN clause, 4-10
where allowed, 4-9
explicit cursors, 6-6
expressions
Boolean, 2-28
CASE, 2-31
how evaluated, 2-23
parentheses in, 2-24
syntax, 13-72
EXTEND collection method, 5-27
extensibility, 8-3
external references, 8-50
how resolved, 8-51
external routines, 8-63

F
FALSE value, 2-9
features, new, xxxi
FETCH statement, 6-9, 6-23
syntax, 13-83
fetching
across commits, 6-51
bulk, 5-42
Fibonacci sequence, 8-58
fields, 5-46
file I/O, 9-17
FIRST collection method, 5-25
flag, PLSQL_V2_COMPATIBILITY, 6-62
FLOAT subtype, 3-4
FOR loops, 4-13
cursor, 6-13
dynamic range, 4-15
iteration scheme, 4-13
loop counter, 4-13
nested, 4-16
FOR UPDATE clause, 6-7
restriction on, 6-20
when to use, 6-50
FORALL statement, 5-34
syntax, 13-88
using with BULK COLLECT clause, 5-45
formal parameters, 6-8
format
masks, 3-26
of functions, 8-6
of packaged subprograms, 8-11
of procedures, 8-3
forward declarations, 8-10
when needed, 8-10, 8-61
forward references, 2-17
forward type definitions, 10-32
%FOUND cursor attribute, 6-35, 6-39
functions, 8-1, 8-6
body, 8-7
built-in, 2-35
call, 8-8
parameter, 8-6
parts, 8-7
RETURN clause, 8-7
specification, 8-7
syntax, 13-90

G

gigabyte, 3-14
GOTO statement, 4-17
  label, 4-17
  misuse, 4-19
  restriction, 7-17
  syntax, 13-97
GROUP BY clause, 6-3
GROUPING aggregate function, 6-2
guess, 3-9

H

handlers, exception, 7-2
handling exceptions, 7-1
  raised in declaration, 7-16
  raised in handler, 7-17
  using OTHERS handler, 7-15
handling of nulls, 2-33
hidden declarations, 9-3
hint, DETERMINISTIC, 8-7
hint, NOCOPY, 8-17
host arrays
  bulk binds, 5-45
hypertext markup language (HTML), 9-18
hypertext transfer protocol (HTTP), 9-18

I

identifiers
  forming, 2-5
  maximum length, 2-5
  quoted, 2-6
  scope rules, 2-19
IF statement, 4-2
ELSE clause, 4-3
ELSIF clause, 4-4
syntax, 13-99
THEN clause, 4-3
implicit cursors, 6-11
attributes, 6-38
implicit datatype conversion, 3-24
  effect on performance, 12-3
implicit declarations
  cursor FOR loop record, 6-13
  FOR loop counter, 4-15
IN comparison operator, 2-28, 6-5
IN OUT parameter mode, 8-16
IN parameter mode, 8-14
incomplete object types, 10-32
index-by tables
  versus nested tables, 5-3
infinite loops, 4-9
information hiding, 1-16, 9-5
inheritance
  and overloading, 8-28
  in PL/SQL, 10-14
initialization
  collection, 5-8
  object, 10-25
  package, 9-8
  record, 5-48
  using DEFAULT, 2-12
  variable, 2-22
  when required, 2-13
INSERT statement
  RETURNING clause, 12-12
  syntax, 13-102
instances, 10-4
INT subtype, 3-4
INTEGER subtype, 3-4
inter-language calls, 8-63
INTERSECT set operator, 6-6
INTERVAL DAY TO SECOND datatype, 3-20
INTERVAL YEAR TO MONTH datatype, 3-19
INTO clause, 6-23
INTO list, 6-9
INVALID_CURSOR exception, 7-5
INVALID_NUMBER exception, 7-5
invoker rights, 8-47
  advantages, 8-48
  versus definer rights, 8-47
IS DANGLING predicate, 10-35
IS NULL comparison operator, 2-27, 6-5
IS OF predicate, 10-14
%ISOPEN cursor attribute, 6-35, 6-39
iteration
scheme, 4-13
versus recursion, 8-62
iterative control, 4-9

J
joins, 8-60

L
labels
block, 2-21
GOTO statement, 4-17
loop, 4-11
large object (LOB) datatypes, 3-13
LAST collection method, 5-25
LEVEL pseudocolumn, 6-4
lexical units, 2-2
LIKE comparison operator, 2-27, 6-5
LIMIT clause, 5-43
LIMIT collection method, 5-25
limitations, PL/SQL, E-1
literals, 2-7
   Boolean, 2-9
caracter, 2-8
datetime, 2-9
numeric, 2-7
string, 2-9
syntax, 13-105
LOB (large object) datatypes, 3-13
lob locators, 3-13
local subprograms, 1-19
locator variables, 7-22
LOCK TABLE statement, 6-51
   syntax, 13-108
locks, 6-43
   modes, 6-43
   overriding, 6-49
   using FOR UPDATE clause, 6-50
logical rowids, 3-7
LOGIN_DENIED exception, 7-5
LONG datatype, 3-6
   maximum length, 3-6
LONG RAW datatype, 3-6
   converting, 3-27
   maximum length, 3-6
LOOP statement, 4-9
   forms, 4-9
   syntax, 13-110
loops
   counters, 4-13
   kinds, 4-9
   labels, 4-11
maintainability, 8-3
makefile
   for native compilation of PL/SQL
      procedures, 12-14
map methods, 10-10
MAX aggregate function, 6-2
maximum precision, 3-4
maximum size
   CHAR value, 3-5
   identifier, 2-5
   LOB, 3-13
   LONG RAW value, 3-6
   LONG value, 3-6
   NCHAR value, 3-12
   NVARCHAR2 value, 3-12
   Oracle error message, 7-18
   RAW value, 3-7
   VARCHAR2 value, 3-9
membership test, 2-28
MERGE statement
   syntax, 13-117
method calls, chaining, 10-29
methods
   collection, 5-23
   COUNT, 5-24
   DELETE, 5-28, 13-24
   EXISTS, 5-24
   EXTEND, 5-27
   FIRST, 5-25
   LAST, 5-25
   LIMIT, 5-25
   map, 10-10
NEXT, 5-26
object, 10-3, 10-8
order, 10-10
PRIOR, 5-26
TRIM, 5-27
MIN aggregate function, 6-2
MINUS set operator, 6-6
mixed notation, 8-13
modes, parameter
IN, 8-14
IN OUT, 8-16
OUT, 8-14
modularity, 1-12, 8-3, 9-5
multilevel collections, 5-21
multi-line comments, 2-10
mutual recursion, 8-61

name resolution, 2-18, D-1
named notation, 8-13
names
cursor, 6-7
qualified, 2-17
savepoint, 6-47
variable, 2-18
naming conventions, 2-17
national character datatypes, 3-10
national character set, 3-11
National Language Support (NLS), 3-11
native dynamic SQL. See dynamic SQL
native execution
compiling PL/SQL procedures for, 12-14
NATURAL subtype, 3-3
NATURALN subtype, 3-3
NCHAR datatype, 3-11
NCLOB datatype, 3-15
nested collections, 5-21
nested cursors, 6-41
nested tables
manipulating, 5-13
versus index-by tables, 5-3
nesting
block, 1-3
FOR loop, 4-16
object, 10-7
record, 5-47
network traffic
reducing, 1-22
new features, xxxi
NEXT collection method, 5-26
NEXTVAL pseudocolumn, 6-3
nibble, 3-27
NLS (National Language Support), 3-11
NO_DATA_FOUND exception, 7-5
NOCOPY compiler hint, 8-17
restrictions on, 8-19
non-blank-padding semantics, B-3
NOT logical operator
treatment of nulls, 2-33
NOT NULL constraint
effect on %TYPE declaration, 2-14
effect on performance, 12-4
restriction, 6-7, 8-4
using in collection declaration, 5-6
using in field declaration, 5-49
using in variable declaration, 2-13
NOT_LOGGED_ON exception, 7-5
notation
mixed, 8-13
positional versus named, 8-13
%NOTFOUND cursor attribute, 6-35
NOWAIT parameter, 6-50
NVARCHAR2 datatype, 3-12
NVL function
treatment of nulls, 2-35
null handling, 2-33
in dynamic SQL, 11-17
NULL statement, 4-21
syntax, 13-117, 13-119
using in a procedure, 8-4
nullity, 2-27
NUMBER datatype, 3-3
numeric literals, 2-7
NUMERIC subtype, 3-4

O

object attributes, 10-3, 10-7
accessing, 10-27
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowed datatypes</td>
<td>10-7</td>
</tr>
<tr>
<td>maximum number</td>
<td>10-7</td>
</tr>
<tr>
<td>object constructors</td>
<td>10-28</td>
</tr>
<tr>
<td>calling</td>
<td>10-28</td>
</tr>
<tr>
<td>passing parameters to</td>
<td>10-28</td>
</tr>
<tr>
<td>object methods</td>
<td>10-3, 10-8</td>
</tr>
<tr>
<td>calling</td>
<td>10-28</td>
</tr>
<tr>
<td>object tables</td>
<td>10-33</td>
</tr>
<tr>
<td>object types</td>
<td>10-1, 10-3</td>
</tr>
<tr>
<td>advantages</td>
<td>10-5</td>
</tr>
<tr>
<td>defining</td>
<td>10-13</td>
</tr>
<tr>
<td>examples</td>
<td>10-13</td>
</tr>
<tr>
<td>structure</td>
<td>10-5</td>
</tr>
<tr>
<td>syntax</td>
<td>13-120</td>
</tr>
<tr>
<td>object-oriented programming</td>
<td>10-1</td>
</tr>
<tr>
<td>objects</td>
<td>10-4</td>
</tr>
<tr>
<td>declaring</td>
<td>10-25</td>
</tr>
<tr>
<td>initializing</td>
<td>10-25</td>
</tr>
<tr>
<td>manipulating</td>
<td>10-32</td>
</tr>
<tr>
<td>sharing</td>
<td>10-29</td>
</tr>
<tr>
<td>OPEN statement</td>
<td>6-7</td>
</tr>
<tr>
<td>syntax</td>
<td>13-129</td>
</tr>
<tr>
<td>OPEN-FOR statement</td>
<td>6-19</td>
</tr>
<tr>
<td>syntax</td>
<td>13-131</td>
</tr>
<tr>
<td>OPEN-FOR-USING statement</td>
<td>11-8</td>
</tr>
<tr>
<td>syntax</td>
<td>13-134</td>
</tr>
<tr>
<td>operators</td>
<td></td>
</tr>
<tr>
<td>comparison</td>
<td>2-26</td>
</tr>
<tr>
<td>precedence</td>
<td>2-24</td>
</tr>
<tr>
<td>relational</td>
<td>2-27</td>
</tr>
<tr>
<td>option, PARALLEL_ENABLE</td>
<td>8-6</td>
</tr>
<tr>
<td>OR keyword</td>
<td>7-16</td>
</tr>
<tr>
<td>order methods</td>
<td>10-10</td>
</tr>
<tr>
<td>order of evaluation</td>
<td>2-24, 2-25</td>
</tr>
<tr>
<td>OTHERS exception handler</td>
<td>7-2, 7-15</td>
</tr>
<tr>
<td>OUT parameter mode</td>
<td>8-14</td>
</tr>
<tr>
<td>overloading</td>
<td>8-23</td>
</tr>
<tr>
<td>and inheritance</td>
<td>8-28</td>
</tr>
<tr>
<td>object method</td>
<td>10-10</td>
</tr>
<tr>
<td>packaged subprogram</td>
<td>9-15</td>
</tr>
<tr>
<td>restrictions</td>
<td>8-24</td>
</tr>
<tr>
<td>using subtypes</td>
<td>8-25</td>
</tr>
<tr>
<td>overriding methods</td>
<td>10-14</td>
</tr>
<tr>
<td>packaged cursors</td>
<td>6-12</td>
</tr>
<tr>
<td>packaged subprograms</td>
<td>1-19, 8-11</td>
</tr>
<tr>
<td>calling</td>
<td>9-7</td>
</tr>
<tr>
<td>overloading</td>
<td>9-15</td>
</tr>
<tr>
<td>packages</td>
<td>9-1, 9-2</td>
</tr>
<tr>
<td>advantages</td>
<td>9-5</td>
</tr>
<tr>
<td>bodiless</td>
<td>9-6</td>
</tr>
<tr>
<td>body</td>
<td>9-2</td>
</tr>
<tr>
<td>initializing</td>
<td>9-8</td>
</tr>
<tr>
<td>private versus public objects</td>
<td>9-14</td>
</tr>
<tr>
<td>product-specific</td>
<td>9-16</td>
</tr>
<tr>
<td>referencing</td>
<td>9-7</td>
</tr>
<tr>
<td>scope</td>
<td>9-6</td>
</tr>
<tr>
<td>serially reusable</td>
<td>12-6</td>
</tr>
<tr>
<td>specification</td>
<td>9-2</td>
</tr>
<tr>
<td>syntax</td>
<td>13-137</td>
</tr>
<tr>
<td>parallel execution of table functions</td>
<td>8-40</td>
</tr>
<tr>
<td>PARALLEL_ENABLE option</td>
<td>8-6</td>
</tr>
<tr>
<td>parameter aliasing</td>
<td>8-21</td>
</tr>
<tr>
<td>parameter passing</td>
<td></td>
</tr>
<tr>
<td>by reference</td>
<td>8-21</td>
</tr>
<tr>
<td>by value</td>
<td>8-21</td>
</tr>
<tr>
<td>in dynamic SQL</td>
<td>11-7</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>actual versus formal</td>
<td>8-12</td>
</tr>
<tr>
<td>cursor</td>
<td>6-8</td>
</tr>
<tr>
<td>default values</td>
<td>8-19</td>
</tr>
<tr>
<td>modes</td>
<td>8-14</td>
</tr>
<tr>
<td>SELF</td>
<td>10-8</td>
</tr>
<tr>
<td>parentheses</td>
<td>2-24</td>
</tr>
<tr>
<td>PARTITION BY clause</td>
<td></td>
</tr>
<tr>
<td>of CREATE FUNCTION statement</td>
<td>8-42</td>
</tr>
<tr>
<td>partitioned data</td>
<td></td>
</tr>
<tr>
<td>and table functions</td>
<td>8-40</td>
</tr>
<tr>
<td>pattern matching</td>
<td>2-27</td>
</tr>
<tr>
<td>performance</td>
<td>1-21</td>
</tr>
<tr>
<td>physical rowids</td>
<td>3-7</td>
</tr>
<tr>
<td>pipe</td>
<td>9-17</td>
</tr>
<tr>
<td>PIPE ROW statement</td>
<td></td>
</tr>
<tr>
<td>for returning rows incrementally</td>
<td>8-35</td>
</tr>
<tr>
<td>pipelining</td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>8-30</td>
</tr>
<tr>
<td>placeholders</td>
<td>11-2</td>
</tr>
</tbody>
</table>
RAISE statement, 7-11
   syntax, 13-148
   using in exception handler, 7-15
raise_application_error procedure, 7-9
raising an exception, 7-11
range operator, 4-13
RAW datatype, 3-7
   converting, 3-27
      maximum length, 3-7
read consistency, 6-44
READ ONLY parameter, 6-49
readability, 2-2, 4-21
read-only transaction, 6-48
REAL subtype, 3-4
RECORD datatype, 5-46
records, 5-46
   %ROWTYPE, 6-13
   assigning, 5-51
   comparing, 5-53
   declaring, 5-48
   defining, 5-46
   implicit declaration, 6-13
   initializing, 5-48
   manipulating, 5-53
   nesting, 5-47
   referencing, 5-49
   syntax, 13-150
recursion, 8-58
   infinite, 8-58
   mutual, 8-61
      terminating condition, 8-58
      versus iteration, 8-62
REF CURSOR datatype, 6-16
   defining, 6-17
REF CURSOR variables
   as parameters to table functions, 8-38
   predefined SYS_REFCURSOR type, 8-38
REF function, 10-35
REF type modifier, 10-31
reference datatypes, 3-2
references, external, 8-50
refs, 10-30
   dangling, 10-35
   declaring, 10-31
   dereferencing, 10-36
relational operators, 2-27
remote access indicator, 2-17
REPEAT UNTIL structure
   mimicking, 4-12
REPLACE function
   treatment of nulls, 2-35
   reraising an exception, 7-14
reserved words, F-1
   misuse of, 2-6
   using as quoted identifier, 2-7
resolution, name, 2-18, D-1
RESTRICT_REFERENCES pragma, 8-9
   syntax, 13-154
   using with autonomous functions, 6-61
   using with dynamic SQL, 11-19
restricted rowids, 3-8
result set, 1-5, 6-7
RETURN clause
   cursor, 6-12
   function, 8-7
RETURN statement, 8-8
   syntax, 13-157
return type, 6-17, 8-25
return value, function, 8-7
RETURNING clause, 10-38, 12-12
reusability, 8-3
reusable packages, 12-6
REVERSE reserved word, 4-13
rollback segments, 6-44
ROLLBACK statement, 6-45
   effect on savepoints, 6-47
   syntax, 13-159
rollbacks
   implicit, 6-47
      of FORALL statement, 5-35
      statement-level, 6-46
routines, external, 8-63
row locks, 6-50
row operators, 6-6
%ROWCOUNT cursor attribute, 6-36, 6-39
ROWID datatype, 3-7
ROWID pseudocolumn, 6-4
rowids, 3-7
extended, 3-8
guess, 3-9
logical, 3-7
physical, 3-7
restricted, 3-8
universal, 3-7
ROWIDTOCHAR function, 6-4
ROWNUM pseudocolumn, 6-4
%ROWTYPE attribute, 2-14
   syntax, 13-161
ROWTYPE_MISMATCH exception, 7-6
RPC (remote procedure call), 7-12
RTRIM function
   using to insert data, B-4
rules, purity, 8-9
run-time errors, 7-1

S

sample database tables
   DEPT table, xxviii
   EMP table, xxviii
sample programs, A-1
savepoint names
   reusing, 6-47
SAVEPOINT statement, 6-46
   syntax, 13-163
scalar datatypes, 3-2
scale
   specifying, 3-4
   schemes, iteration, 4-13
   scientific notation, 2-8
scope, 2-19
   collection, 5-8
   cursor, 6-7
   cursor parameter, 6-7
   definition, 2-19
   exception, 7-7
   identifier, 2-19
   loop counter, 4-15
   package, 9-6
searched CASE expression, 2-32
SELECT INTO statement
   syntax, 13-164
selector, 2-31

SELF parameter, 10-8
semantics
   assignment, B-2
   blank-padding, B-3
   CHAR versus VARCHAR2, B-1
   non-blank-padding, B-3
   string comparison, B-2
separators, 2-3
sequence, 6-3
sequential control, 4-17
serially reusable packages, 12-6
SERIALLY_REUSABLE pragma, 12-6
   syntax, 13-169
Server Pages, PL/SQL, 8-64
session, 6-43
session-specific variables, 9-11
set operators, 6-6
SET TRANSACTION statement, 6-48
   syntax, 13-171
short-circuit evaluation, 2-26
side effects, 8-14
   controlling, 8-9
significant characters, 2-5
SIGNTYPE subtype, 3-3
simple symbols, 2-3
single-line comments, 2-10
size limit, varray, 5-5
SMALLINT subtype, 3-4
snapshots, 6-44
SOME comparison operator, 6-5
spaces
   where allowed, 2-2
spaghetti code, 4-17
sparse collections, 5-3
specification
   call, 9-3
   cursor, 6-12
   function, 8-7
   method, 10-8
   object, 10-5
   package, 9-6
   procedure, 8-4
SQL
   comparison operators, 6-5
data manipulation statements, 6-2
dynamic, 11-2
pseudocolumns, 6-3
row operators, 6-6
set operators, 6-6
support in PL/SQL, 1-20
SQL cursor
  syntax, 13-173
SQLCODE function, 7-18
  syntax, 13-176
SQLERRM function, 7-18
  syntax, 13-178
stack, 10-16
standalone subprograms, 1-19
START WITH clause, 6-4
statement terminator, 13-16
statement-level rollbacks, 6-46
statements, PL/SQL
  assignment, 13-4
  CASE, 13-18
  CLOSE, 6-11, 6-24, 13-21
  COMMIT, 13-36
  DELETE, 13-57
dynamic SQL, 11-2
  EXECUTE IMMEDIATE, 11-3
  EXIT, 13-70
  FETCH, 6-9, 6-23, 13-83
  FORALL, 5-34
  GOTO, 13-97
  IF, 13-99
  INSERT, 13-102
  LOCK TABLE, 13-108
  LOOP, 13-110
  MERGE, 13-117
  NULL, 13-117, 13-119
  OPEN, 6-7, 6-129
  OPEN-FOR, 6-19, 13-131
  OPEN-FOR-USING, 11-8
  RAISE, 13-148
  RETURN, 13-157
  ROLLBACK, 13-159
  SAVEPOINT, 13-163
  SELECT INTO, 13-164
  SET TRANSACTION, 13-171
  UPDATE, 13-182
  STDDEV aggregate function, 6-2
STEP clause
  mimicking, 4-14
stepwise refinement, 1-2
STORAGE_ERROR exception, 7-6
  when raised, 8-58
store tables, 5-5
stored subprograms, 1-18
streaming, data
  definition, 8-45
string comparison semantics, B-2
string literals, 2-9
STRING subtype, 3-10
structure theorem, 4-2
stubs, 4-21, 8-3
subprograms, 8-2
  advantages, 8-3
  declaring, 8-10
  how calls are resolved, 8-26
  local, 1-19
  overloading, 8-23
  packaged, 1-19, 8-11
  parts, 8-2
  procedure versus function, 8-6
  recursive, 8-58
  standalone, 1-19
  stored, 1-18
subquery, 6-11
SUBSCRIPT_BEYOND_COUNT exception, 7-6
SUBSCRIPT_OUTSIDE_LIMIT exception, 7-6
substitutability of object types, 8-28
SUBSTR function, 7-19
subtypes, 3-3, 3-21, 10-14
  CHARACTER, 3-6
  compatibility, 3-23
  constrained versus unconstrained, 3-22
  DEC, 3-4
  DECIMAL, 3-4
  defining, 3-22
  DOUBLE PRECISION, 3-4
  FLOAT, 3-4
  INT, 3-4
  INTEGER, 3-4
  NATURAL, 3-3
  NATURALN, 3-3
  NUMERIC, 3-4
overloading, 8-25
POSITIVE, 3-3
POSITIVEN, 3-3
REAL, 3-4
SIGNTYPE, 3-3
SMALLINT, 3-4
STRING, 3-10
VARCHAR, 3-10
SUM aggregate function, 6-2
supertypes, 10-14
support for SQL, 6-2
symbols
compound, 2-4
simple, 2-3
syntax
definition, 13-1
diagram reading, 13-3
SYS_REFCURSOR type, 8-38
system-specific Oracle documentation
   PL/SQL wrapper, 12-15

T
TABLE datatype, 5-2
table functions, 8-30
doing transformations with, 8-33
   parallel execution, 8-40
   pipelined, 8-32
   querying, 8-36
TABLE operator, 5-18
tabs, 2-2
terminating condition, 8-58
terminator, statement, 2-3
ternary operators, 2-23
THEN clause, 4-3
TIMEOUT_ONRESOURCE exception, 7-6
TIMESTAMP datatype, 3-17
TIMESTAMP WITH LOCAL TIME ZONE
datatype, 3-18
TIMESTAMP WITH TIME ZONE datatype, 3-17
TOO_MANY_ROWS exception, 7-6
top-down design, 1-16
trailing blanks
   how handled, B-4
transactions, 6-2
autonomous, 6-53
committing, 6-44
context, 6-56
distributed, 6-44
ending properly, 6-48
processing, 6-2, 6-43
read-only, 6-48
rolling back, 6-45
visibility, 6-56
TREAT operator, 10-14
triggers, 1-19
   autonomous, 6-59
TRIM collection method, 5-27
TRUE value, 2-9
%TYPE attribute, 2-13
   syntax, 13-180
type definitions
   collection, 5-5
   forward, 10-32
   RECORD, 5-46
   REF CURSOR, 6-17
type evolution
   in PL/SQL, 10-12
type inheritance
   in PL/SQL, 10-14

U
unary operators, 2-23
underscores, 2-5
unhandled exceptions, 7-12, 7-19
uninitialized object
   how treated, 10-26
UNION ALL set operator, 6-6
UNION set operator, 6-6
universal rowids, 3-7
UPDATE statement
   RETURNING clause, 12-12
   syntax, 13-182
URL (uniform resource locator), 9-18
UROWID datatype, 3-7
user session, 6-43
user-defined exceptions, 7-7
user-defined records, 5-46
user-defined subtypes, 3-21
USING clause, 11-3, 13-67
UTF8 character encoding, 3-11
UTL_FILE package, 9-17
UTL_HTTP package, 9-18

V
VALUE function, 10-34
VALUE_ERROR exception, 7-6
VARCHAR subtype, 3-10
VARCHAR2
  datatype, 3-9
  semantics, B-1
variables
  assigning values, 2-22
  declaring, 2-11
  initializing, 2-22
  session-specific, 9-11
  syntax, 13-38
VARIANCE aggregate function, 6-2
VARRAY datatype, 5-4
varrays
  size limit, 5-5
visibility
  of package contents, 9-3
  transaction, 6-56
  versus scope, 2-19

W
WHEN clause, 4-10, 7-15
WHILE loop, 4-11
wildcards, 2-27
words, reserved, F-1
work areas, query, 6-16
Wrap Utility, C-1
  input and output files, C-3
  running, C-2

Z
ZERO_DIVIDE exception, 7-6