PL/SQL

User's Guide and Reference

10*g* Release 1 (10.1)

Part No. B10807-01

December 2003

ORACLE®

PL/SQL User's Guide and Reference, 10*g* Release 1 (10.1)

Part No. B10807-01

Primary Author: John Russell

Contributors: Shashaanka Agrawal, Cailein Barclay, Dmitri Bronnikov, Sharon Castledine, Thomas Chang, Ravindra Dani, Chandrasekharan Iyer, Susan Kotsovolos, Neil Le, Warren Li, Bryn Llewellyn, Chris Racicot, Murali Vemulapati, Guhan Viswanathan, Minghui Yang

Contents

2 Fundamentals of the PL/SQL Language

3 PL/SQL Datatypes

4 Using PL/SQL Control Structures

5 Using PL/SQL Collections and Records

6 Performing SQL Operations from PL/SQL

7 Performing SQL Operations with Native Dynamic SQL

8 Using PL/SQL Subprograms

9 Using PL/SQL Packages

10 Handling PL/SQL Errors

11 Tuning PL/SQL Applications for Performance

12 Using PL/SQL Object Types

13 PL/SQL Language Elements

Send Us Your Comments

PL/SQL User's Guide and Reference, 10*g* Release 1 (10.1)

Part No. B10807-01

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation 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 Manager
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

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

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

Preface

This guide explains the concepts behind the PL/SQL language and shows, with examples, how to use various language features.

This preface contains these topics:

Audience

PL/SQL, Oracle's procedural extension of SQL, is an advanced fourth-generation programming language (4GL). It offers software-engineering features such as data encapsulation, overloading, collection types, exceptions, and information hiding. PL/SQL also supports rapid prototyping and development through tight integration with SQL and the Oracle database.

Anyone developing PL/SQL-based applications for Oracle should read this book. This book is intended for programmers, systems analysts, project managers, database administrators, and others who need to automate database operations. People developing applications in other languages can also produce mixed-language applications with parts written in PL/SQL.

To use this guide effectively, you need a working knowledge of the Oracle database, the SQL language, and basic programming constructs such as `IF-THEN` comparisons, loops, and procedures and functions.

How This Book Is Organized

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

Getting Started with PL/SQL

Chapter 1, "Overview of PL/SQL"

Summarizes the main features of PL/SQL and their advantages. Introduces the basic concepts behind PL/SQL and the general appearance of PL/SQL programs.

Chapter 2, "Fundamentals of the PL/SQL Language"

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

Chapter 3, "PL/SQL Datatypes"

Discusses PL/SQL's predefined datatypes, which include integer, floating-point, character, Boolean, date, collection, reference, and LOB types. Also discusses user-defined subtypes and data conversion.

Chapter 4, "Using PL/SQL Control Structures"

Shows how to control the flow of execution through a PL/SQL program. Describes conditional, iterative, and sequential control, with control structures such as `IF-THEN-ELSE`, `CASE`, and `WHILE-LOOP`.

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

Discusses the composite datatypes `TABLE`, `VARRAY`, and `RECORD`. You learn how to reference and manipulate whole collections of data and group data of different types together.

Database Programming with PL/SQL

Chapter 6, "Performing SQL Operations from PL/SQL"

Shows how PL/SQL supports the SQL commands, functions, and operators for manipulating Oracle data. Also shows how to process queries and transactions.

Chapter 7, "Performing SQL Operations with Native Dynamic SQL"

Shows how to build SQL statements and queries at run time.

Software Engineering with PL/SQL

Chapter 8, "Using PL/SQL Subprograms"

Shows how to write and call procedures, functions. It discusses related topics such as parameters, overloading, and different privilege models for subprograms.

Chapter 9, "Using PL/SQL Packages"

Shows how to bundle related PL/SQL types, items, and subprograms into a package. Packages define APIs that can be reused by many applications.

Chapter 10, "Handling PL/SQL Errors"

Shows how to detect and handle PL/SQL errors using exceptions and handlers.

Chapter 11, "Tuning PL/SQL Applications for Performance"

Shows how to improve performance for PL/SQL-based applications.

Chapter 12, "Using PL/SQL Object Types"

Introduces object-oriented programming based on object types. You learn how to write object methods and manipulate objects through PL/SQL.

PL/SQL Language Reference

Chapter 13, "PL/SQL Language Elements"

Shows the syntax of statements, parameters, and other PL/SQL language elements. Also includes usage notes and short examples.

Appendixes

Appendix A, "Sample PL/SQL Programs"

Provides several PL/SQL programs to guide you in writing your own. The sample programs illustrate important concepts and features.

Appendix B, "Understanding CHAR and VARCHAR2 Semantics in PL/SQL"

Explains the subtle but important semantic differences between the `CHAR` and `VARCHAR2` base types.

Appendix C, "Obfuscating Source Code with the PL/SQL Wrap Utility"

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, "How PL/SQL Resolves Identifier Names"

Explains how PL/SQL resolves references to names in potentially ambiguous SQL and procedural statements.

Appendix E, "PL/SQL Program Limits"

Explains the compile-time and runtime limits imposed by PL/SQL .

Appendix F, "List of PL/SQL Reserved Words"

Lists the words that are reserved for use by PL/SQL.

Appendix G, "Frequently Asked Questions About PL/SQL"

Provides tips and answers to some of the most common PL/SQL questions.

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 *Oracle Database Application Developer's Guide - Fundamentals*

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

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

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

Printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

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://otn.oracle.com/membership/`

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://otn.oracle.com/documentation/`

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.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	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.	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.
`lowercase monospace (fixed-width font)`	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. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	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.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

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:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (digits [ , precision ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	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: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
. . .	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;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

--	A double hyphen begins a single-line comment, which extends to the end of a line.	`--`
/* */	A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines.	`/* */`

Sample Database Tables

Some programming examples in this guide use tables and other objects from the HR schema of the sample database. These tables, such as EMPLOYEES and DEPARTMENTS,

are more extensive and realistic than the EMP and DEPT tables used in previous releases.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. 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/

Accessibility of Code Examples in Documentation 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.

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

Reading the Syntax Diagrams

To understand the syntax of a PL/SQL statement, trace through its syntax diagram, reading from left to right and top to bottom.

The diagrams represent Backus-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.

What's New in PL/SQL?

This section describes new features of PL/SQL release 10*g*, and provides pointers to additional information.

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

- New Features in PL/SQL for Oracle Database 10g
- New Features in PL/SQL for Oracle9i

New Features in PL/SQL for Oracle Database 10g

Release 1 (10.1)

Improved Performance

PL/SQL performance is improved across the board. Most improvements are automatic, with no action required from you.

Global optimization of PL/SQL code is controlled by the `PLSQL_OPTIMIZE_LEVEL` initialization parameter. The default optimization level improves performance for a broad range of PL/SQL operations. Most users should never need to change the default optimization level.

Performance improvements include better integer performance, reuse of expression values, simplification of branching code, better performance for some library calls, and elimination of dead code.

The new datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` can improve performance in number-crunching applications, such as processing scientific data.

Native compilation is easier and more integrated, with fewer initialization parameters to set, less compiler configuration, the object code stored in the database, and compatibility with Oracle Real Application Clusters environments.

The `FORALL` statement can handle associate arrays and nested tables with deleted elements. You can now use this performance construct in more situations than before, and avoid the need to copy elements from one collection to another.

Enhancements to PL/SQL Native Compilation

This feature now requires less setup and maintenance.

A package body and its spec do not need to be compiled with the same setting for native compilation. For example, a package body can be compiled natively while the package spec is compiled interpreted, or vice versa.

Natively compiled subprograms are stored in the database, and the corresponding shared libraries are extracted automatically as needed. You do not need to worry about backing up the shared libraries, cleaning up old shared libraries, or what happens if a shared library is deleted accidentally.

The initialization parameters and command setup for native compilation have been simplified. The only required parameter is `PLSQL_NATIVE_LIBRARY_DIR`. The parameters related to the compiler, linker, and make utility have been obsoleted. The file that controls compilation is now a command file showing the commands and options for compiling and linking, rather than a makefile. Any errors that occur during native compilation are reflected in the `USER_ERRORS` dictionary view and by the SQL*Plus command `SHOW ERRORS`.

Native compilation is turned on and off by a separate initialization parameter, `PLSQL_CODE_TYPE`, rather than being one of several options in the `PLSQL_COMPILER_FLAGS` parameter, which is now deprecated.

See Also:

- "Compiling PL/SQL Code for Native Execution" on page 11-22

FORALL Support for Non-Consecutive Indexes

You can use the `INDICES OF` and `VALUES OF` clauses with the `FORALL` statement to iterate over non-consecutive index values. For example, you can delete elements from a nested table, and still use that nested table in a `FORALL` statement.

See Also:

- "Using the FORALL Statement" on page 11-8

New IEEE Floating-Point Types

New datatypes `BINARY_FLOAT` and `BINARY_DOUBLE` represent floating-point numbers in IEEE 754 format. These types are useful for scientific computation where you exchange data with other programs and languages that use the IEEE 754 standard for floating-point. Because many computer systems support IEEE 754 floating-point operations through native processor instructions, these types are efficient for intensive computations involving floating-point data.

Support for these types includes numeric literals such as `1.0f` and `3.141d`, arithmetic operations including square root and remainder, exception handling, and special values such as not-a-number (NaN) and infinity.

The rules for overloading subprograms are enhanced, so that you can write math libraries with different versions of the same function operating on `PLS_INTEGER`, `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE` parameters.

See Also:

- "PL/SQL Number Types" on page 3-2

Improved Overloading

You can now overload subprograms that accept different kinds of numeric arguments, to write math libraries with specialized versions of each subprogram for different datatypes.

See Also:

- "Guidelines for Overloading with Numeric Types" on page 8-11

Nested Table Enhancements

Nested tables defined in PL/SQL have many more operations than previously. You can compare nested tables for equality, test whether an element is a member of a nested table, test whether one nested table is a subset of another, perform set operations such as union and intersection, and much more.

See Also:

- "Assigning Collections" on page 5-13
- "Comparing Collections" on page 5-16

Compile-Time Warnings

Oracle can issue warnings when you compile subprograms that produce ambiguous results or use inefficient constructs. You can selectively enable and disable these warnings through the `PLSQL_WARNINGS` initialization parameter and the `DBMS_WARNING` package.

See Also:

- "Overview of PL/SQL Compile-Time Warnings" on page 10-17

Quoting Mechanism for String Literals

Instead of doubling each single quote inside a string literal, you can specify your own delimiter character for the literal, and then use single quotes inside the string.

See Also:

- "String Literals" on page 2-6

Implicit Conversion Between CLOB and NCLOB

You can implicitly convert from CLOB to NCLOB or from NCLOB to CLOB. Because this can be an expensive operation, it might help maintainability to continue using the TO_CLOB and TO_NCLOB functions.

Regular Expressions

If you are familiar with UNIX-style regular expressions, you can use them while performing queries and string manipulations. You use the `REGEXP_LIKE` operator in SQL queries, and the `REGEXP_INSTR`, `REGEXP_REPLACE`, and `REGEXP_SUBSTR` functions anywhere you would use `INSTR`, `REPLACE`, and `SUBSTR`.

See Also:

- "Summary of PL/SQL Built-In Functions" on page 2-28
- "How Can I Use Regular Expressions with PL/SQL?" on page G-1

Flashback Query Functions

The functions `SCN_TO_TIMESTAMP` and `TIMESTAMP_TO_SCN` let you translate between a date and time, and the system change number that represents the database state at a point in time.

See Also:

- "SCN_TO_TIMESTAMP Function" on page 13-122
- "TIMESTAMP_TO_SCN Function" on page 13-138

New Features in PL/SQL for Oracle9i

Release 2 (9.2)

- **Insert/update/select of entire PL/SQL records**

  You can now insert into or update a SQL table by specifying a PL/SQL record variable, rather than specifying each record attribute separately. You can also select entire rows into a PL/SQL table of records, rather than using a separate PL/SQL table for each SQL column.

  See Also:

  - "Inserting PL/SQL Records into the Database" on page 5-36
  - "Updating the Database with PL/SQL Record Values" on page 5-36
  - "Querying Data into Collections of Records" on page 5-38

- **Associative arrays**

  You can create collections that are indexed by `VARCHAR2` values, providing features similar to hash tables in Perl and other languages.

  See Also:

  - "Understanding Associative Arrays (Index-By Tables)" on page 5-3

- **User-defined constructors**

  You can now override the system default constructor for an object type with your own function.

  See Also:

  - "Defining Object Constructors" on page 12-13

- **Enhancements to UTL_FILE package**

  `UTL_FILE` contains several new functions that let you perform general file-management operations from PL/SQL.

  See Also:

  - *PL/SQL Packages and Types Reference*

- **TREAT function for object types**

  You can dynamically choose the level of type inheritance to use when calling object methods. That is, you can reference an object type that inherits from several levels of parent types, and call a method from a specific parent type. This function is similar to the SQL function of the same name.

See Also:

- *Oracle Database SQL Reference*

- **Better linking in online documentation**

  Many of the cross-references from this book to other books have been made more specific, so that they link to a particular place within another book rather than to the table of contents. Because this is an ongoing project, not all links are improved in this edition. If you are reading a printed copy of this book, you can find the online equivalent at `http://otn.oracle.com/documentation/`, with full search capability.

Release 1 (9.0.1)

- **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 *Oracle Database Upgrade Guide* 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 2-24
  - "Using the CASE Statement" on page 4-3
  - "CASE Statement" on page 13-14

- **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 12-10
  - "How Overloading Works with Inheritance" on page 8-13

- **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 12-9

- **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:

  - "PL/SQL Date, Time, and Interval Types" on page 3-12
  - "Datetime and Interval Arithmetic" on page 3-15
  - "Datetime Literals" on page 2-6

- **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 11-22

- **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.

  See Also:

  - "PL/SQL Character and String Types" on page 3-4
  - "PL/SQL National Character Types" on page 3-8

- **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:

  - "Setting Up Transformation Pipelines with Table Functions" on page 11-28
  - "Using Cursor Expressions" on page 6-27

- **Multilevel Collections**

  You can nest the collection types, for example to create a `VARRAY` of PL/SQL tables, a `VARRAY` of `VARRAY`s, 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 `RAW`s. Conversions between LOBs and other types are much simpler, particularly when converting from `LONG` to LOB types.

  See Also: "PL/SQL LOB Types" on page 3-10

- **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 DML Statements and Queries (FORALL, BULK COLLECT)" on page 11-7
  - "Using Bulk Dynamic SQL" on page 7-6
  - "EXECUTE IMMEDIATE Statement" on page 13-47

- **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 13-84 for a brief discussion and example
  - *Oracle Database SQL Reference* for detailed information

1

Overview of PL/SQL

The limits of my language mean the limits of my world. —Ludwig Wittgenstein

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

This chapter contains these topics:

> **See Also:** Access additional information and code samples for PL/SQL on the Oracle Technology Network, at `http://otn.oracle.com/tech/pl_sql/` .

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

Tight Integration with SQL

The PL/SQL language is tightly integrated with SQL. You do not have to translate between SQL and PL/SQL datatypes: a `NUMBER` or `VARCHAR2` column in the database is stored in a `NUMBER` or `VARCHAR2` variable in PL/SQL. This integration saves you both learning time and processing time. Special PL/SQL language features let you work with table columns and rows without specifying the datatypes, saving on maintenance work when the table definitions change.

Running a SQL query and processing the result set is as easy in PL/SQL as opening a text file and processing each line in popular scripting languages.

Using PL/SQL to access metadata about database objects and handle database error conditions, you can write utility programs for database administration that are reliable and produce readable output about the success of each operation.

Many database features, such as triggers and object types, make use of PL/SQL. You can write the bodies of triggers and methods for object types in PL/SQL.

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.

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. This extensive SQL support lets you manipulate Oracle data flexibly and safely. Also, PL/SQL fully supports SQL datatypes, reducing the need to convert data passed between your applications and the database.

PL/SQL also supports dynamic SQL, a 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 at run time, without knowing details such as table names and `WHERE` clauses in advance.

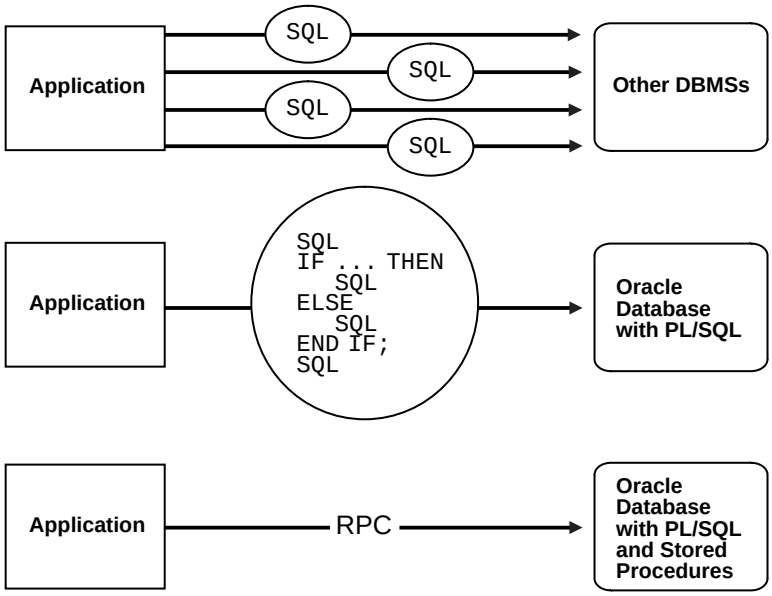
Better Performance

Without PL/SQL, Oracle must process SQL statements one at a time. Programs that issue many SQL statements require multiple calls to the database, resulting in significant network and performance overhead.

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

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

Figure 1–1 PL/SQL Boosts Performance



Higher Productivity

PL/SQL extends tools such as Oracle Forms and Oracle Reports. With PL/SQL in these tools, you can use familiar language constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger, instead of multiple trigger steps, macros, or user exits.

PL/SQL is the same in all environments. Once you learn PL/SQL with one Oracle tool, you can transfer your knowledge to other tools.

Full Portability

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

Tight Security

PL/SQL stored procedures move application code from the client to the server, where you can protect it from tampering, hide the internal details, and restrict who has access. For example, you can grant users access to a procedure that updates a table, but not grant them access to the table itself or to the text of the `UPDATE` statement.

Triggers written in PL/SQL can control or record changes to data, making sure that all changes obey your business rules.

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. This direct mapping helps your programs better reflect the world they are trying to simulate.

Understanding the Main Features of PL/SQL

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

You can control program flow with statements like `IF` and `LOOP`. As with other procedural programming languages, you can declare variables, define procedures and functions, and trap runtime errors.

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

Block Structure

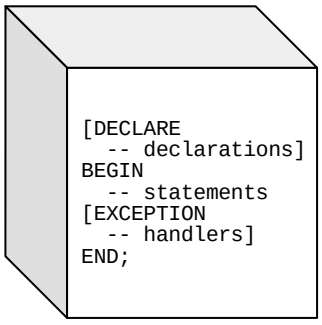
The basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can be nested inside one another.

A block groups related declarations and statements. You can place declarations close to where they are used, even inside a large subprogram. The declarations are local to the block and cease to exist when the block completes, helping to avoid cluttered namespaces for variables and procedures.

As Figure 1–2 shows, a PL/SQL block has three parts: a declarative part, an executable part, and an exception-handling part that deals with error conditions. Only the executable part is required.

First comes the declarative part, where you define types, variables, and similar items. These items are manipulated in the executable part. Exceptions raised during execution can be dealt with in the exception-handling part.

Figure 1–2 Block Structure



You can nest blocks in the executable and exception-handling parts of a PL/SQL block or subprogram but not in the declarative part. You can define local subprograms in the declarative part of any block. You can 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. You must declare a constant or variable before referencing it in any other statements.

Declaring Variables

Variables can have any SQL datatype, such as CHAR, DATE, or NUMBER, or a PL/SQL-only datatype, such as BOOLEAN or PLS_INTEGER. For example, assume that you want to declare 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 salary * 0.10 INTO bonus FROM employees WHERE employee_id = 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, and doing the assignment inside the subprogram. The following example passes a variable to a subprogram, and the subprogram updates the variable:

```
DECLARE
   my_sal REAL(7,2);
   PROCEDURE adjust_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. No further assignments to the constant are allowed. The following example declares a constant:

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

Processing Queries with PL/SQL

Processing a SQL query with PL/SQL is like processing files with other languages. For example, a Perl program opens a file, reads the file contents, processes each line, then closes the file. In the same way, a PL/SQL program issues a query and processes the rows from the result set:

```
FOR someone IN (SELECT * FROM employees)
LOOP
  DBMS_OUTPUT.PUT_LINE('First name = ' || someone.first_name);
  DBMS_OUTPUT.PUT_LINE('Last name = ' || someone.last_name);
END LOOP;
```

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

Declaring PL/SQL Variables

As part of the declaration for each PL/SQL variable, you declare its datatype. Usually, this datatype is one of the types shared between PL/SQL and SQL, such as `NUMBER` or `VARCHAR2(length)`. For easier maintenance of code that interacts with the database, you can also use the special qualifiers `%TYPE` and `%ROWTYPE` to declare variables that hold table columns or table rows.

%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.

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

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

```
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:

```
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

```
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.

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.

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.

```
-- 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:

```
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;
```

Writing Reusable PL/SQL Code

PL/SQL lets you break an application down into manageable, well-defined modules. PL/SQL meets this need with *program units*, which include blocks, subprograms, and packages. You can reuse program units by loading them into the database as triggers, stored procedures, and stored functions.

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:

```
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, a database object that is a step above regular stored procedures. The packages defines a simple, clear, interface to a set of related procedures and types.

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

The following example packages two employment procedures:

```
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;
```

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

Packages are stored in the database, where they can be shared by many applications. Calling a packaged subprogram for the first time loads the whole package and caches it in memory, saving on disk I/O for subsequent calls. Thus, packages enhance reuse and improve performance in a multi-user, multi-application environment.

Data Abstraction

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

Collections

PL/SQL collection types let you declare high-level datatypes similar to arrays, sets, and hash tables found in other languages. In PL/SQL, array types are known as varrays (short for variable-size arrays), set types are known as nested tables, and hash table types are known as associative arrays. Each kind of collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection.

To reference an element, use subscript notation with parentheses. For example, the following call references the fifth element in the nested table (of type `Staff`) returned by function `new_hires`:

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

Records

Records are composite data structures whose fields can have different datatypes. You can use records to hold related items and pass them to subprograms with a single parameter.

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

Consider the following example:

```
DECLARE
   TYPE TimeRec IS RECORD (hours SMALLINT, minutes SMALLINT);
   TYPE MeetingTyp IS RECORD (
      date_held DATE,
      duration  TimeRec,  -- nested record
      location  VARCHAR2(20),
      purpose   VARCHAR2(50));
```

Object Types

PL/SQL supports object-oriented programming through 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 known as attributes. The functions and procedures that manipulate the attributes are known as methods.

Object types reduce complexity by breaking down a large system into logical entities. This lets you create software components that are modular, maintainable, and reusable.

Object-type definitions, and the code for the methods, are stored in the database. Instances of these object types can be stored in tables or used as variables inside PL/SQL code.

```
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
);
```

Error Handling

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

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

You can declare exceptions of your own, for conditions that you decide are errors, or to correspond to database errors that normally result in ORA- error messages. When you detect a user-defined error condition, you execute a `RAISE` statement. The following example computes the bonus earned by a salesperson. The bonus is based on salary and commission. If the commission is null, you raise the exception `comm_missing`.

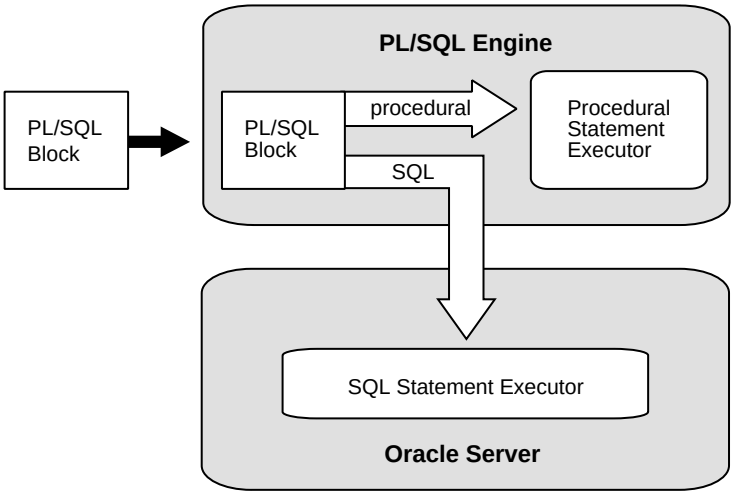
```
DECLARE
   comm_missing EXCEPTION;  -- declare exception
BEGIN
   IF commission IS NULL THEN
      RAISE comm_missing;  -- raise exception
   END IF;
   bonus := (salary * 0.10) + (commission * 0.15);
EXCEPTION
   WHEN comm_missing THEN ... -- process the exception
```

PL/SQL Architecture

The PL/SQL compilation and run-time system is 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.

In either environment, the PL/SQL engine accepts as input any valid PL/SQL block or subprogram. Figure 1–3 shows the PL/SQL engine processing an anonymous block. The PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine in the Oracle database.

Figure 1–3 PL/SQL Engine



In the Oracle Database Server

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

Anonymous Blocks

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

Stored Subprograms

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

Stored subprograms defined within a package are known as packaged subprograms. Those defined independently are called standalone subprograms.

Subprograms nested inside other subprograms or within a PL/SQL block are known as local subprograms, which cannot be referenced by other applications and exist only inside the enclosing block.

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

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

For example, you might call the standalone procedure `create_dept` from SQL*Plus as follows:

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

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

Database Triggers

A database trigger is a stored subprogram associated with a database table, view, or event. The trigger can be called once, when some event occurs, or many times, once for each row affected by an `INSERT`, `UPDATE`, or `DELETE` statement. The trigger can be called after the event, to record it or take some followup action. Or, the trigger can be called before the event to prevent erroneous operations or fix new data so that it conforms to business rules. For example, the following table-level trigger fires whenever salaries in the `emp` table are updated:

```
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 *Oracle Database Application Developer's Guide - Fundamentals*.

In Oracle Tools

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

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

2

Fundamentals of the PL/SQL Language

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 detailed aspects of the language. Like other programming languages, PL/SQL has a character set, reserved words, punctuation, datatypes, and fixed syntax rules.

This chapter contains these topics:

Character Set

You write a PL/SQL program as lines of text using a specific set of characters:

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

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

Lexical Units

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

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

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

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

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

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

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

```
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.

Symbol	Meaning
+	addition operator
%	attribute indicator
'	character string delimiter
.	component selector
/	division operator
(	expression or list delimiter
)	expression or list delimiter
:	host variable indicator
,	item separator
*	multiplication operator
"	quoted identifier delimiter
=	relational operator
<	relational operator
>	relational operator
@	remote access indicator
;	statement terminator
-	subtraction/negation operator

Symbol	Meaning
:=	assignment operator
=>	association operator
||	concatenation operator
**	exponentiation operator
<<	label delimiter (begin)
>>	label delimiter (end)
/*	multi-line comment delimiter (begin)
*/	multi-line comment delimiter (end)
..	range operator
<>	relational operator
!=	relational operator
~=	relational operator
^=	relational operator
<=	relational operator
>=	relational operator
--	single-line comment indicator

Identifiers

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

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

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

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

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

```
money$$$tree
SN##
try_again_
```

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

```
lastname
```

```
LastName  -- same as lastname
LASTNAME  -- same as lastname and LastName
```

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

```
lastname
last_name
```

Identifiers should be descriptive. 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. For example, the words `BEGIN` and `END` are reserved. Trying to redefine a reserved word causes a compilation error. Instead, you can embed reserved words as part of a longer identifier:

```
DECLARE
   -- end BOOLEAN;          -- not allowed; causes compilation error
   end_of_game BOOLEAN;  -- allowed
BEGIN
   NULL;
END;
/
```

Often, reserved words are written in upper case for readability. 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.

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:

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

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

```
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   1.0E-7   3.14159e0   -1E38   -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` is multiplied (the double asterisk (`**`) is the exponentiation operator):

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

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

```
5E-3 = 5 * 10**-3 = 5 * 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
END;
/
```

Real literals can also use the trailing letters `f` and `d` to specify the types `BINARY_FLOAT` and `BINARY_DOUBLE`, respectively:

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

Character Literals

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

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

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

String Literals

A character value can be represented by an identifier or explicitly written as a string literal, which is a sequence of zero or more characters enclosed by single quotes. 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`.

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

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

Doubling the quotation marks within a complicated literal, particularly one that represents a SQL statement, can be tricky. You can also use the following notation to define your own delimiter characters for the literal. You choose a character that is not present in the string, and then do not need to escape other single quotation marks inside the literal:

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

-- To use delimiters [, {, <, and (, pair them with ], }, >, and ).
-- Here we pass a string literal representing a SQL statement
-- to a subprogram, without doubling the quotation marks around
-- 'INVALID'.
func_call(q'[select index_name from user_indexes where status = 'INVALID']');

-- For NCHAR and NVARCHAR2 literals, use the prefix nq instead of q.
where_clause := nq'#where col_value like '%é'#';
```

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 `YEAR TO MONTH` or `DAY TO SECOND`. For example, `current_timestamp - current_timestamp` produces a value of type `INTERVAL DAY TO SECOND` by default. You can specify the type of the interval using the formats:

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

For details on the syntax for the date and time types, see the *Oracle Database SQL Reference*. For examples of performing date/time arithmetic, see *Oracle Database 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 (`--`) anywhere on a line and extend to the end of the line. A few examples follow:

```
DECLARE
   howmany NUMBER;
BEGIN
-- begin processing
   SELECT count(*) INTO howmany FROM user_objects
      WHERE object_type = 'TABLE'; -- Check number of tables
   howmany := howmany * 2;         -- Compute some other value
END;
/
```

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

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

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

Multi-line Comments

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

```
DECLARE
   some_condition BOOLEAN;
   pi NUMBER := 3.1415926; radius NUMBER := 15; area NUMBER;
BEGIN
   /* Perform some simple tests and assignments */
   IF 2 + 2 = 4 THEN
      some_condition := TRUE; /* We expect this THEN to always be done */
```

```
   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:

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

Restrictions on Comments

You cannot nest comments.

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

Declarations

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

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

A couple of examples follow:

```
DECLARE
   birthday  DATE;
   emp_count SMALLINT := 0;
```

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

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

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

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

To declare a constant, put the keyword CONSTANT before the type specifier:

```
DECLARE
   credit_limit CONSTANT REAL := 5000.00;
   max_days_in_year CONSTANT INTEGER := 366;
   urban_legend CONSTANT BOOLEAN := FALSE;
BEGIN
   NULL;
END;
/
```

This declaration names a constant of type REAL and assigns an unchangeable value of 5000 to the constant. A constant must be initialized in its declaration. Otherwise, you get a compilation error.

Using DEFAULT

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

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

can be rewritten as follows:

```
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. For example:

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

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

Using NOT NULL

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

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

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

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

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

Using the %TYPE Attribute

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

```
DECLARE
   credit NUMBER(7,2);
   debit  credit%TYPE;
```

```
   name VARCHAR2(20) := 'JoHn SmItH';
-- If we increase the length of NAME, the other variables
-- become longer too.
   upper_name name%TYPE := UPPER(name);
   lower_name name%TYPE := LOWER(name);
   init_name name%TYPE := INITCAP(name);
BEGIN
   NULL;
END;
/
```

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. A %TYPE declaration can also include an initialization clause.

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

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

When you use `table_name.column_name.TYPE` to declare a variable, you do not need to know the actual datatype, and attributes such as precision, scale, and length. If the database definition of the column changes, the datatype of the variable changes accordingly at run time.

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

```
DECLARE
   my_empno employees.employee_id%TYPE;
BEGIN
   my_empno := NULL;  -- this works
END;
/
```

Using the %ROWTYPE Attribute

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:

```
DECLARE
-- %ROWTYPE can include all the columns in a table...
   emp_rec employees%ROWTYPE;
-- ...or a subset of the columns, based on a cursor.
   CURSOR c1 IS
      SELECT department_id, department_name FROM departments;
   dept_rec c1%ROWTYPE;
-- Could even make a %ROWTYPE with columns from multiple tables.
   CURSOR c2 IS
      SELECT employee_id, email, employees.manager_id, location_id
```

```
      FROM employees, departments
      WHERE employees.department_id = departments.department_id;
   join_rec c2%ROWTYPE;
BEGIN
-- We know EMP_REC can hold a row from the EMPLOYEES table.
   SELECT * INTO emp_rec FROM employees WHERE ROWNUM < 2;
-- We can refer to the fields of EMP_REC using column names
-- from the EMPLOYEES table.
   IF emp_rec.department_id = 20 AND emp_rec.last_name = 'JOHNSON' THEN
      emp_rec.salary := emp_rec.salary * 1.15;
   END IF;
END;
/
```

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.

Aggregate Assignment

Although a %ROWTYPE declaration cannot include an initialization clause, there are ways to assign values to all fields in a record at once. You can assign one record to another if their declarations refer to the same table or cursor. For example, the following assignment is allowed:

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

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

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

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

Using Aliases

Select-list items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases. The following example uses an alias called complete_name to represent the concatenation of two columns:

```
BEGIN
-- We assign an alias (COMPLETE_NAME) to the expression value, because
-- it has no column name.
```

```
   FOR item IN
   (
      SELECT first_name || ' ' || last_name complete_name
      FROM employees WHERE ROWNUM < 11
   )
   LOOP
-- Now we can refer to the field in the record using this alias.
      dbms_output.put_line('Employee name: ' || item.complete_name);
   END LOOP;
END;
/
```

Restrictions on Declarations

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

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

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

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

PL/SQL Naming Conventions

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

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

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

Synonyms

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

Scoping

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

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

For the scoping rules that apply to identifiers, see "Scope and Visibility of PL/SQL Identifiers" on page 2-14.

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:

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

Name Resolution

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

To avoid ambiguity, add a prefix to the names of local variables and formal parameters, or use a block label to qualify references.

```
CREATE TABLE employees2 AS SELECT last_name FROM employees;

<<MAIN>>
DECLARE
   last_name VARCHAR2(10) := 'King';
   my_last_name VARCHAR2(10) := 'King';
BEGIN
-- Deletes everyone, because both LAST_NAMEs refer to the column
   DELETE FROM employees2 WHERE last_name = last_name;
   dbms_output.put_line('Deleted ' || SQL%ROWCOUNT || ' rows.');
   ROLLBACK;
```

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

DROP TABLE employees2;
```

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

```
DECLARE
   FUNCTION dept_name (department_id IN NUMBER)
      RETURN departments.department_name%TYPE
   IS
      department_name departments.department_name%TYPE;
   BEGIN
-- DEPT_NAME.DEPARTMENT_NAME specifies the local variable
-- instead of the table column
      SELECT department_name INTO dept_name.department_name
         FROM departments
         WHERE department_id = dept_name.department_id;
      RETURN department_name;
   END;

BEGIN
   FOR item IN (SELECT department_id FROM departments)
   LOOP
      dbms_output.put_line('Department: ' || dept_name(item.department_id));
   END LOOP;
END;
/
```

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

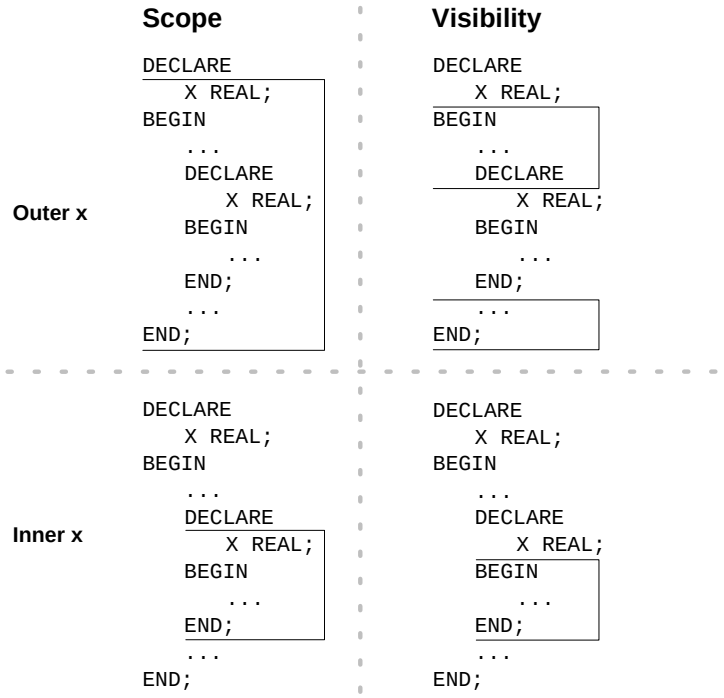
Scope and Visibility of PL/SQL Identifiers

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

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

Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks. The two items represented by the identifier are distinct, and any change in one does not affect the other. However, a block cannot reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.

Figure 2–1 Scope and Visibility



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.

```
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:

```
<<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.

Assigning Values to Variables

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

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

Unless you expressly initialize a variable, its value is undefined (NULL).

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

```
DECLARE
   counter INTEGER;
BEGIN
-- COUNTER is initially NULL, so 'COUNTER + 1' is also null.
   counter := counter + 1;
   IF counter IS NULL THEN
      dbms_output.put_line('Sure enough, COUNTER is NULL not 1.');
   END IF;
END;
/
```

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

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

Assigning Boolean Values

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. You can assign these literal values, or expressions such as comparisons using relational operators.

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

Assigning a SQL Query Result to a PL/SQL Variable

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. For example:

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

Because SQL does not have a Boolean type, you cannot select column values into a Boolean variable.

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

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

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

Operators with higher precedence are applied first. In the 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.

```
5 + 12 / 4
12 / 4 + 5
```

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

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

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

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

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

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

Logical Operators

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

Table 2–2 Logic Truth Table

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

Table 2–2 (Cont.) Logic Truth Table

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

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

NOT NULL returns NULL, because nulls are indeterminate. Be careful to avoid unexpected results in expressions involving nulls; see "Handling Null Values in Comparisons and Conditional Statements" on page 2-25.

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:

```
DECLARE
   on_hand  INTEGER := 0;
   on_order INTEGER := 100;
BEGIN
-- Does not cause divide-by-zero error; evaluation stops after 1st expr.
   IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
      dbms_output.put_line('There are no more widgets left!');
   END IF;
END;
/
```

When the value of on_hand is zero, the left operand yields TRUE, so PL/SQL does not evaluate the right operand. If PL/SQL evaluated both operands before applying the OR operator, the right operand would cause a *division by zero* error.

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 some examples of comparisons for different types:

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

Relational Operators

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

IS NULL Operator

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

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

LIKE Operator

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

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

```
ename LIKE 'J%S_N'
```

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

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

BETWEEN Operator

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

```
45 BETWEEN 38 AND 44
```

IN Operator

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

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

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

```
value NOT IN set
```

yield FALSE if the set contains a null.

Concatenation Operator

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

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

returns the following value:

```
'suitcase'
```

If 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

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

the following expression is true:

```
number1 > number2
```

Boolean Character Expressions

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

For example, given the assignments

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

the following expression is true:

```
string1 > string2
```

By setting the initialization parameter NLS_COMP=ANSI, you can make comparisons use the collating sequence identified by the NLS_SORT initialization parameter. A **collating sequence** is an internal ordering of the character set in which a range of numeric codes represents the individual characters. One character value is greater than another if its internal numeric value is larger. Each language might have different rules about where such characters occur in the collating sequence. For example, an accented letter might be sorted differently depending on the database character set, even though the binary value is the same in each case.

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

There are semantic differences between the CHAR and VARCHAR2 base types that come into play when you compare character values. For more information, see 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 "PL/SQL Character and String Types" on page 3-4.

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. For example, the following `IF` condition might not yield `TRUE`:

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

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

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

  The debugged version follows:

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

- A Boolean variable is itself either true or false. You can just use the variable in a conditional test, rather than comparing it to the literal values `TRUE` and `FALSE`. For example, the following loops are all equivalent:

  ```
  DECLARE
     done BOOLEAN ;
  BEGIN
  -- Each WHILE loop is equivalent
     done := FALSE;
     WHILE done = FALSE
     LOOP
        done := TRUE;
     END LOOP;

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

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

```
/
```

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

CASE Expressions

A CASE expression selects a result from one or more alternatives, and returns the result. Although it contains a block that might stretch over several lines, it really is an expression that forms part of a larger statement, such as an assignment or a procedure call.

The CASE expression uses a **selector**, an expression whose value determines which alternative to return. 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 evaluated. The first WHEN clause that matches the value of the selector determines the result value, and subsequent WHEN clauses are not evaluated. For example:

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

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

An alternative to the CASE expression is the CASE statement, where each WHEN clause can be an entire PL/SQL block. For details, see "Using the CASE Statement" on page 4-3.

Searched CASE Expression

PL/SQL also provides a *searched* CASE expression, which lets you test different conditions instead of comparing a single expression to various values. It 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, so you can test different variables or multiple conditions in a single WHEN clause. For example:

```
DECLARE
   grade CHAR(1) := 'B';
   appraisal VARCHAR2(120);
   id NUMBER := 8429862;
   attendance NUMBER := 150;
   min_days CONSTANT NUMBER := 200;
FUNCTION attends_this_school(id NUMBER) RETURN BOOLEAN IS
   BEGIN RETURN TRUE; END;
BEGIN
   appraisal :=
      CASE
         WHEN attends_this_school(id) = FALSE THEN 'N/A - Student not enrolled'
-- Have to put this condition early to detect
-- good students with bad attendance
         WHEN grade = 'F' OR attendance < min_days THEN 'Poor (poor performance or
bad attendance)'
         WHEN grade = 'A' THEN 'Excellent'
         WHEN grade = 'B' THEN 'Very Good'
         WHEN grade = 'C' THEN 'Good'
         WHEN grade = 'D' THEN 'Fair'
         ELSE 'No such grade'
      END;
   dbms_output.put_line('Result for student ' || id ||
      ' is ' || appraisal);
END;
/
```

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

Handling Null Values in Comparisons and Conditional Statements

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

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

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

In the 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.

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

NULLs and the NOT Operator

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

```
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.

NULLs and Zero-Length Strings

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

```
DECLARE
```

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

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

```
IF my_string IS NULL THEN ...
```

NULLs and the Concatenation Operator

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

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

returns the following value:

```
'applesauce'
```

NULLs as Arguments to Built-In Functions

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

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

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

The function `NVL` returns the value of its second argument if its first argument is null. In the following example, if the column specified in the query is null, the function returns the value -1 to signify a non-existent employee in the output:

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

The function REPLACE returns the value of its first argument if its second argument is null, whether the optional third argument is present or not. For example, the following call to REPLACE does not make any change to the value of OLD_STRING:

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

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

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

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

Summary of PL/SQL 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 *Oracle Database 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

Error	Number	Character	Conversion	Date	Obj Ref	Misc
SQLCODE	ABS	ASCII	CHARTOROWID	ADD_MONTHS	DEREF	BFILENAME
SQLERRM	ACOS	ASCIISTR	CONVERT	CURRENT_DATE	REF	COALESCE
	ASIN	CHR	HEXTORAW	CURRENT_TIME	TREAT	DECODE
	ATAN	COMPOSE	RAWTOHEX	CURRENT_TIMESTAMP	VALUE	DUMP
	ATAN2	CONCAT	RAWTONHEX	DBTIMEZONE		EMPTY_BLOB
	BITAND	DECOMPOSE	ROWIDTOCHAR	EXTRACT		EMPTY_CLOB
	CEIL	INITCAP	TO_BINARY_DOUBLE	FROM_TZ		GREATEST
	COS	INSTR	TO_BLOB	LAST_DAY		LEAST
	COSH	INSTR2	TO_BINARY_FLOAT	LOCALTIMESTAMP		NANVL
	EXP	INSTR4	TO_CHAR	MONTHS_BETWEEN		NLS_CHARSET_DECL_LEN
	FLOOR	INSTRB	TO_CLOB	NEW_TIME		NLS_CHARSET_ID
	LN	INSTRC	TO_DATE	NEXT_DAY		NLS_CHARSET_NAME
	LOG	LENGTH	TO_MULTI_BYTE	NUMTODSINTERVAL		NULLIF
	MOD	LENGTH2	TO_NCHAR	NUMTOYMINTERVAL		NVL
	POWER	LENGTH4	TO_NCLOB	ROUND		SYS_CONTEXT
	REMAINDER	LENGTHB	TO_NUMBER	SESSIONTIMEZONE		SYS_GUID
	ROUND	LENGTHC	TO_SINGLE_BYTE	SYS_EXTRACT_UTC		UID
	SIGN	LOWER		SYSDATE		USER
	SIN	LPAD		SYSTIMESTAMP		USERENV
	SINH	LTRIM		TO_DSINTERVAL		VSIZE
	SQRT	NCHR		TO_TIME		
	TAN	NLS_INITCAP		TO_TIME_TZ		
	TANH	NLS_LOWER		TO_TIMESTAMP		
	TRUNC	NLSSORT		TO_TIMESTAMP_TZ		
		NLS_UPPER		TO_YMINTERVAL		
		REGEXP_INSTR		TRUNC		
		REGEXP_LIKE		TZ_OFFSET		
		REGEXP_REPLACE				
		REGEXP_SUBSTR				
		REPLACE				
		RPAD				
		RTRIM				
		SOUNDEX				
		SUBSTR				
		SUBSTR2				
		SUBSTR4				
		SUBSTRB				
		SUBSTRC				
		TRANSLATE				
		TRIM				
		UNISTR				
		UPPER				

3

PL/SQL Datatypes

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 many predefined datatypes. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and large object (LOB) types. PL/SQL also 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 contains these topics:

Overview of Predefined PL/SQL Datatypes

A *scalar* type has no internal components. It holds a single value, such as a number or character string.

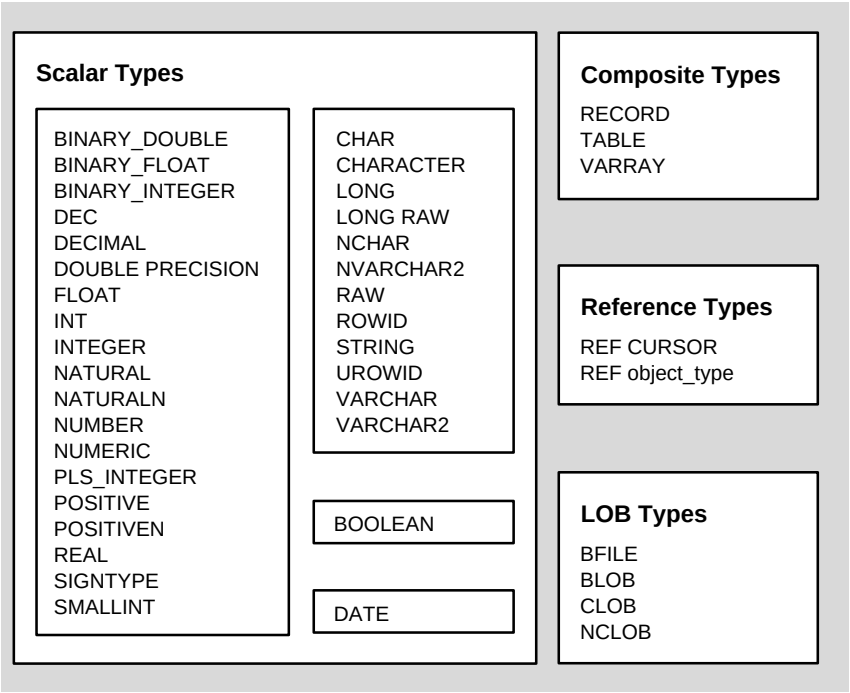
A *composite* type has internal components that can be manipulated individually, such as the elements of an array.

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, such as text blocks or graphic images, that are stored separately from other database data.

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

Figure 3–1 Built-in Datatypes



PL/SQL 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.

`BINARY_INTEGER` values require less storage than `NUMBER` values. Arithmetic operations on `BINARY_INTEGER` values are also faster than `NUMBER` arithmetic. `BINARY_INTEGER` and `PLS_INTEGER` both have these advantages. Because `PLS_INTEGER` was faster in earlier releases, you might use it instead of `BINARY_INTEGER` in code that will run on older databases.

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.

BINARY_FLOAT and BINARY_DOUBLE

Single-precision and double-precision IEEE 754-format single-precision floating-point numbers. These types are used primarily for high-speed scientific computation. For usage information, see "Writing Computation-Intensive Programs in PL/SQL" on page 11-19. For information about writing math libraries that accept different numeric types, see "Guidelines for Overloading with Numeric Types" on page 8-11.

Literals of these types end with `f` (for `BINARY_FLOAT`) or `d` (for `BINARY_DOUBLE`). For example, `2.07f` or `3.000094d`.

Computations involving these types produce special values that you need to check for, rather than raising exceptions. To help deal with overflow, underflow, and other conditions that can occur with these numbers, you can use several special predefined constants: `BINARY_FLOAT_NAN`, `BINARY_FLOAT_INFINITY`, `BINARY_FLOAT_MAX_NORMAL`, `BINARY_FLOAT_MIN_NORMAL`, `BINARY_FLOAT_MAX_SUBNORMAL`, `BINARY_FLOAT_MIN_SUBNORMAL`, and corresponding names starting with `BINARY_DOUBLE`. The constants for `NaN` ("not a number") and infinity are also defined by SQL; the others are PL/SQL-only.

NUMBER

You use the `NUMBER` datatype to store fixed-point or floating-point numbers. Its magnitude range is 1E-130 .. 10E125. 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:

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

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

```
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.

PL/SQL Character and String 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, 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 Oracle9*i*, 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 "PL/SQL LOB Types" on page 3-10 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 *Oracle Database 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.

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

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

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 UROWID variable with the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement to identify the latest row fetched from a cursor. See "Fetching Across Commits" on page 6-34.

A physical rowid can have either of two formats. The 10-byte *extended rowid* format supports tablespace-relative block addresses and can identify rows in partitioned and non-partitioned tables. The 6-byte *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

```
SQL> SELECT rowid, ename FROM emp WHERE empno = 7788;
```

might return the following row:

```
ROWID              ENAME
------------------ ----------
AAAAqcAABAAADFNAAH SCOTT
```

The format, OOOOOOFFFBBBBBBRRR, has four parts:

- 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.
- FFF: The file number (AAB in the example) identifies the data file that contains the row. File numbers are unique within a database.
- BBBBBB: The block number (AAADFN in the example) identifies the data block that contains the row. Because block numbers are relative to their data file, not their tablespace, 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 *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:

```
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, 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. It is a good idea to use VARCHAR2 rather than VARCHAR.

PL/SQL 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 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 *Oracle Database Globalization 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 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:

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

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.

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

```
my_string NCHAR(100);  -- maximum size is 100 characters
```

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

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:

```
NVARCHAR2(maximum_size)
```

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 (?).

PL/SQL 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 that LOB in another transaction. Likewise, you cannot save a LOB locator during one session, then use it in another session.

Starting in Oracle9*i*, 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 *Oracle Database Application Developer's Guide - Large Objects*.

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.

PL/SQL Boolean Types

PL/SQL has a type for representing Boolean values (true and false). Because SQL does not have an equivalent type, you can use BOOLEAN variables and parameters in PL/SQL contexts but not inside SQL statements or queries.

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. You cannot select or fetch column values into a BOOLEAN variable. Functions called from a SQL

query cannot take any BOOLEAN parameters. Neither can built-in SQL functions such as TO_CHAR; to represent BOOLEAN values in output, you must use IF-THEN or CASE constructs to translate BOOLEAN values into some other type, such as 0 or 1, 'Y' or 'N', 'true' or 'false', and so on.

PL/SQL Date, Time, 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:

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

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

Tips:

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

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

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

You can add and subtract dates. In arithmetic expressions, PL/SQL interprets integer literals as days. For instance, SYSDATE + 1 signifies the same time 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
   checkout := '1999-06-22 07:48:53.275';
   ...
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:

```
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.

```
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:

```
'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:

```
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:

```
DECLARE
   lifetime INTERVAL YEAR(3) TO MONTH;
BEGIN
   lifetime := INTERVAL '101-3' YEAR TO MONTH; -- interval literal
   lifetime := '101-3'; -- implicit conversion from character type
   lifetime := INTERVAL '101' YEAR; -- Can specify just the years
   lifetime := INTERVAL '3' MONTH; -- Can specify just the months
   ...
END;
```

INTERVAL DAY TO SECOND

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:

Operand 1	Operator	Operand 2	Result Type
datetime	+	interval	datetime
datetime	-	interval	datetime
interval	+	datetime	datetime
datetime	-	datetime	interval
interval	+	interval	interval
interval	-	interval	interval

Operand 1	Operator	Operand 2	Result Type
interval	*	numeric	interval
numeric	*	interval	interval
interval	/	numeric	interval

You can also manipulate datetime values using various functions, such as EXTRACT. For a list of such functions, see Table 2–3, " Built-In Functions" on page 2-30.

For further information and examples of datetime arithmetic, see *Oracle Database SQL Reference* and *Oracle Database 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
```

Overview of PL/SQL 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. A subtype does *not* introduce a new type; rather, it 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:

```
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:

```
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. 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.

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

You can constrain a user-defined subtype when declaring variables of that type:

```
DECLARE
   SUBTYPE Accumulator IS NUMBER;
   total Accumulator(7,2);
```

Subtypes can increase reliability by detecting out-of-range values. In 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:

```
DECLARE
   SUBTYPE b1 IS BOOLEAN;
   SUBTYPE b2 IS BOOLEAN;
   finished  b1; -- Different subtypes,
   debugging b2; -- both based on BOOLEAN.
BEGIN
   debugging := finished; -- They can be assigned to each other.
END;
```

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

```
DECLARE
   SUBTYPE Word IS CHAR(15);
   SUBTYPE Text IS VARCHAR2(1500);
   verb     Word;       -- Different subtypes
   sentence Text(150);  -- of types from the same family
BEGIN
   sentence := verb;    -- can be assigned, if not too long.
END;
```

Converting PL/SQL Datatypes

Sometimes it is necessary to convert a value from one datatype to another. For example, to use a `DATE` value in a report, you must convert it to a character string. PL/SQL supports both explicit and implicit (automatic) datatype conversion. To ensure your program does exactly what you expect, use explicit conversions wherever possible.

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 *Oracle Database SQL Reference*.

Using explicit conversions, particularly when passing parameters to subprograms, can avoid unexpected errors or wrong results. For example, the `TO_CHAR` function lets you specify the format for a `DATE` value, rather than relying on language settings in the database. Including an arithmetic expression among strings being concatenated with the `||` operator can produce an error unless you put parentheses or a call to `TO_CHAR` around the entire arithmetic expression.

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. For example, you can pass a numeric literal to a subprogram that expects a string value, and the subprogram receives the string representation of the number.

In the following example, 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`. 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 labels `PLS_INT` and `BIN_INT` represent the types `PLS_INTEGER` and `BINARY_INTEGER` in the table. You cannot use them as abbreviations in code.
- 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.
- You can implicitly convert between `CLOB` and `NCLOB`, but be careful because this can be an expensive operation. To make clear that this conversion is intended, you can 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 *Oracle Database SQL Reference* for details on implicit conversions between different date and time types.

Table 3–1 Implicit Conversions

	BIN_INT	BLOB	CHAR	CLOB	DATE	LONG	NUMBER	PLS_INT	RAW	UROWID	VARCHAR2
BIN_INT			X			X	X	X			X
BLOB									X		
CHAR	X			X	X	X	X	X	X	X	X
CLOB			X								X
DATE			X			X					X
LONG			X						X		X
NUMBER	X		X			X		X			X
PLS_INT	X		X			X	X				X

Table 3–1 (Cont.) Implicit Conversions

	BIN_INT	BLOB	CHAR	CLOB	DATE	LONG	NUMBER	PLS_INT	RAW	UROWID	VARCHAR2
RAW		X	X			X					X
UROWID			X								X
VARCHAR2	X		X	X	X	X	X	X	X	X	

It is your responsibility to ensure that values are convertible. For instance, PL/SQL can convert the CHAR value '02-JUN-92' to a DATE value but cannot convert the CHAR value 'YESTERDAY' to a DATE value. Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value.

Choosing Between Implicit and Explicit Conversion

Relying on implicit datatype conversions is a poor programming practice because they can be slower and the conversion rules might change in later software releases. Implicit conversions are context-sensitive and not always predictable. For best reliability and maintainability, use datatype conversion functions.

DATE Values

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

A conversion is also necessary when you insert a CHAR or VARCHAR2 value into a DATE column. PL/SQL calls the function TO_DATE, which expects the default date format. To insert dates in other formats, 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. Otherwise, PL/SQL raises an exception.

4

Using PL/SQL Control Structures

One ship drives east and another drives west
With the selfsame winds that blow.
'Tis the set of the sails and not the gales
Which tells us the way to go. —Ella Wheeler Wilcox

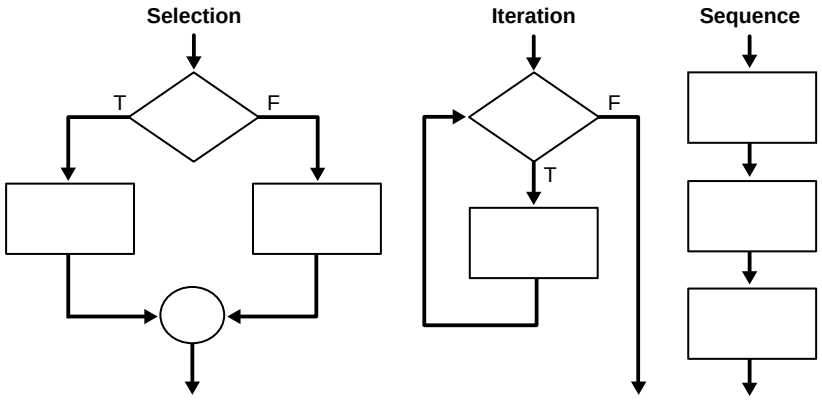
This chapter shows you how to structure the flow of control through a PL/SQL program. PL/SQL provides conditional tests, loops, and branches that let you produce well-structured programs.

This chapter contains these topics:

Overview of PL/SQL Control Structures

Procedural computer programs use the basic control structures shown in Figure 4–1.

Figure 4–1 Control Structures



The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A **condition** is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration

structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Testing Conditions: IF and CASE Statements

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from.

Using the IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF):

```
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.

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

You can place brief IF statements on a single line:

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

Using the IF-THEN-ELSE Statement

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

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

The statements in the ELSE clause are executed only if the condition is false or null. The IF-THEN-ELSE statement ensures that one or the other sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, and the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
   UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
   UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

IF statements can be nested:

```
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;
```

Using the IF-THEN-ELSIF Statement

Sometimes you want to choose between several alternatives. You can use the keyword `ELSIF` (not `ELSEIF` or `ELSE IF`) to introduce additional conditions:

```
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:

```
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.

Using the 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:

```
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:

```
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. 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;
```

There is always a default action, even when you omit the `ELSE` clause. If the `CASE` statement does not match any of the `WHEN` clauses and you omit the `ELSE` clause, PL/SQL raises the predefined exception `CASE_NOT_FOUND`.

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

```
[<<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 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 CASEstatement is the CASE expression, where each WHEN clause is an expression. For details, see "CASE Expressions" on page 2-24.

Searched CASE Statement

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

```
[<<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];
```

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

```
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;
```

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

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

```
ELSE RAISE CASE_NOT_FOUND;
```

Exceptions raised during the execution of a searched CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

Guidelines for PL/SQL Conditional Statements

Avoid clumsy IF statements like those in the following example:

```
IF new_balance < minimum_balance THEN
   overdrawn := TRUE;
```

```
ELSE
   overdrawn := FALSE;
END IF;
...
IF overdrawn = TRUE THEN
   RAISE insufficient_funds;
END IF;
```

The value of a Boolean expression can be assigned directly to a Boolean variable. You can replace the first `IF` statement with a simple assignment:

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

A Boolean variable is itself either true or false. You can simplify the condition in the second `IF` statement:

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

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

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

These statements are logically equivalent, but the second statement makes the logic clearer.

To compare a single expression to multiple values, you can simplify the logic by using a single `CASE` statement instead of an `IF` with several `ELSIF` clauses.

Controlling Loop Iterations: LOOP and EXIT Statements

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

Using the LOOP Statement

The simplest form of `LOOP` statement is the basic loop, which encloses a sequence of statements between the keywords `LOOP` and `END LOOP`, as follows:

```
LOOP
   sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an `EXIT` statement to stop looping and prevent an infinite loop. You can place one or more `EXIT` statements anywhere inside a loop, but not outside a loop. There are two forms of `EXIT` statements: `EXIT` and `EXIT-WHEN`.

Using the EXIT Statement

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:

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

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 8-4.

Using the EXIT-WHEN Statement

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:

```
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. A statement inside the loop must change the value of the condition. In the previous 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.

Labeling a PL/SQL Loop

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:

```
<<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:

```
<<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:

```
<<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.

Using the WHILE-LOOP Statement

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If it is true, the sequence of statements is executed, then control resumes at the top of the loop. If it is false or null, the loop is skipped and control passes to the next statement:

```
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 skipped.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top, so that the sequence of statements is executed at least once. The equivalent in PL/SQL would be:

```
LOOP
   sequence_of_statements
   EXIT WHEN boolean_expression;
END LOOP;
```

To ensure that a WHILE loop executes at least once, use an initialized Boolean variable in the condition, as follows:

```
done := FALSE;
WHILE NOT done LOOP
   sequence_of_statements
   done := boolean_expression;
```

```
END LOOP;
```

A statement inside the loop must assign a new value to the Boolean variable to avoid an infinite loop.

Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator:

```
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;
```

If the lower bound equals the higher bound, the loop body 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. If you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. You still write the range bounds in ascending (not descending) order.

```
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 counter can be read but cannot be changed:

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

Tip: A useful variation of the FOR loop uses a SQL query instead of a range of integers. This technique lets you run a query and process all the rows of the result set with straightforward syntax. For details, see "Querying Data with PL/SQL: Implicit Cursor FOR Loop" on page 6-9.

How PL/SQL Loops Iterate

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

```
j IN -5..5
k IN REVERSE first..last
```

```
step IN 0..TRUNC(high/low) * 2
```

Internally, PL/SQL assigns the values of the bounds to temporary PLS_INTEGER variables, and, if necessary, rounds the values to the nearest integer. The magnitude range of a PLS_INTEGER is -2**31 .. 2**31. If a bound evaluates to a number outside that range, you get a *numeric overflow* error when PL/SQL attempts the assignment.

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 for Loop Bounds

PL/SQL lets you specify the loop range at run time by using variables for bounds:

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

If the lower bound of a loop range evaluates to a larger integer than the upper bound, the loop body is not executed and control passes to the next statement:

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

Scope of the Loop Counter Variable

The loop counter is defined only within the loop. You cannot reference that variable name outside the loop. After the loop exits, the loop counter is undefined:

```
FOR ctr IN 1..10 LOOP
   ...
END LOOP;
sum := ctr - 1;  -- not allowed
```

You do not need to declare the loop counter because it is implicitly declared as a local variable of type INTEGER. It is safest not to use the name of an existing variable, because the local declaration hides any global declaration:

```
DECLARE
   ctr  INTEGER := 3;
BEGIN
   ...
   FOR ctr IN 1..25 LOOP
      ...
```

```
      IF ctr > 10 THEN ...  -- Refers to loop counter
   END LOOP;
-- After the loop, ctr refers to the original variable with value 3.
END;
```

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. To reference the outer loop counter from the inner loop, you use a label and dot notation:

```
<<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 in a FOR Loop

The EXIT statement lets a FOR loop complete early. 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 early from a nested FOR loop. To complete not only the current loop, but also any enclosing loop, label the enclosing loop and use the label in an EXIT statement:

```
<<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 GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement. PL/SQL's exception-handling mechanism is discussed in Chapter 10, "Handling PL/SQL Errors".

Using the GOTO Statement

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
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:

```
DECLARE
   x NUMBER := 0;
BEGIN
   <<increment_x>>
   BEGIN
      x := x + 1;
   END;
   IF x < 10 THEN
      GOTO increment_x;
   END IF;
END;
```

The label end_loop in the following example is not allowed 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>>  -- not allowed
   END LOOP;  -- not an executable statement
END;
```

To correct the previous example, add the NULL statement::

```
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
   my_ename  CHAR(10);
BEGIN
   <<get_name>>
   SELECT ename INTO my_ename FROM emp WHERE ...
   BEGIN
      GOTO get_name;  -- branch to enclosing block
   END;
END;
```

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

Restrictions on the GOTO Statement

Some possible destinations of a GOTO statement are not allowed. Specifically, a GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement, or sub-block. For example, the following GOTO statement is not allowed:

```
BEGIN
   GOTO update_row;  -- can't branch into IF statement
   IF valid THEN
      <<update_row>>
      UPDATE emp SET ...
   END IF;
END;
```

A GOTO statement cannot branch from one IF statement clause to another, or from one CASE statement WHEN clause to another.

A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).

A GOTO statement cannot branch out of a subprogram. To end a subprogram early, you can use the RETURN statement or use GOTO to branch to a place right before the end of the subprogram.

A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

Using the NULL Statement

The NULL statement does nothing, and passes control to the next statement. (Some languages refer to such an instruction as a *no-op*.)

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

```
EXCEPTION
```

```
   WHEN ZERO_DIVIDE THEN
      ROLLBACK;
   WHEN VALUE_ERROR THEN
      INSERT INTO errors VALUES ...
      COMMIT;
   WHEN OTHERS THEN
      NULL;
END;
```

The NULL statement is a handy way to create placeholders and stub procedures. In the following example, the NULL statement lets you compile this procedure, then fill in the real body later:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
BEGIN
   NULL;
END debit_account;
```

5

Using PL/SQL Collections and Records

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. You can model these types in database applications using the PL/SQL datatypes `TABLE` and `VARRAY`, which allow you to declare nested tables, associative arrays, and variable-size arrays. This chapter shows how to reference and manipulate collections of data as local variables. You also learn how the `RECORD` datatype lets you manipulate related values of different types as a logical unit.

This chapter contains these topics:

What Is a Collection?

A **collection** is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other datatypes used in classic programming algorithms. Each element is addressed by a unique subscript.

PL/SQL offers these collection types:

- **Associative arrays**, also known as **index-by tables**, let you look up elements using arbitrary numbers and strings for subscript values. (They are similar to **hash tables** in other programming languages.)

- **Nested tables** hold an arbitrary number of elements. They use sequential numbers as subscripts. You can define equivalent SQL types, allowing nested tables to be stored in database tables and manipulated through SQL.
- **Varrays** (short for variable-size arrays) hold a fixed number of elements (although you can change the number of elements at runtime). They use sequential numbers as subscripts. You can define equivalent SQL types, allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables.

Although collections have only one dimension, you can model multi-dimensional arrays by creating collections whose elements are also collections.

To use collections in an application, you define one or more PL/SQL types, then define variables of those types. You can define collection types in a procedure, function, or package. You can pass collection variables as parameters to stored subprograms.

To look up data that is more complex than single values, you can store PL/SQL records or SQL object types in collections. Nested tables and varrays can also be attributes of object types.

Understanding Nested Tables

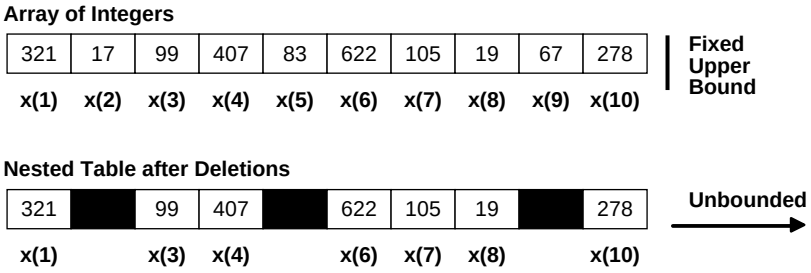
PL/SQL nested tables represent sets of values. You can think of them as one-dimensional arrays with no upper bound. You can model multi-dimensional arrays by creating nested tables whose elements are also nested tables.

Within the database, nested tables are column types that hold sets of values. Oracle stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. That gives you array-like access to individual rows.

Nested tables differ from arrays in two important ways:

1. Nested tables are unbounded, while arrays have a fixed upper bound (see Figure 5–1). The size of a nested table can increase dynamically.
2. Nested tables might not have consecutive subscripts, while arrays are always *dense* (have consecutive subscripts). Initially, nested tables are dense, but they can become *sparse* (have nonconsecutive subscripts). You can delete elements from a nested table using the built-in procedure `DELETE`. The built-in function `NEXT` lets you iterate over all the subscripts of a nested table, even if the sequence has gaps.

Figure 5–1 Array versus Nested Table

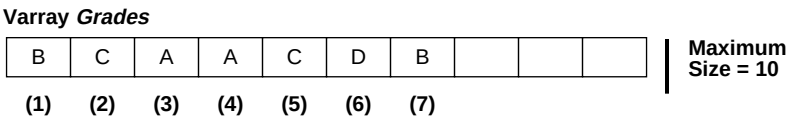


Understanding Varrays

Items of type `VARRAY` are called *varrays*. They let you reference individual elements for array operations, or manipulate the collection as a whole. To reference an element, you

use standard subscripting syntax (see Figure 5–2). For example, `Grade(3)` references the third element in varray `Grades`.

Figure 5–2 Varray of Size 10



A varray has a maximum size, which you specify in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. For example, the current upper bound for varray `Grades` is 7, but you can increase its upper bound to maximum of 10. A varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

Understanding Associative Arrays (Index-By Tables)

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.

Assigning a value using a key for the first time adds that key to the associative array. Subsequent assignments using the same key update the same entry. It is important to choose a key that is unique. For example, key values might come from the primary key of a database table, from a numeric hash function, or from concatenating strings to form a unique string value.

For example, here is the declaration of an associative array type, and two arrays of that type, using keys that are strings:

```
DECLARE
  TYPE population_type IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
  country_population population_type;
  continent_population population_type;
  howmany NUMBER;
  which VARCHAR2(64);
BEGIN
  country_population('Greenland') := 100000; -- Creates new entry
  country_population('Iceland') := 750000;   -- Creates new entry
-- Looks up value associated with a string
  howmany := country_population('Greenland');

  continent_population('Australia') := 30000000;
  continent_population('Antarctica') := 1000; -- Creates new entry
  continent_population('Antarctica') := 1001; -- Replaces previous value

-- Returns 'Antarctica' as that comes first alphabetically.
  which := continent_population.FIRST;
-- Returns 'Australia' as that comes last alphabetically.
  which := continent_population.LAST;
-- Returns the value corresponding to the last key, in this
-- case the population of Australia.
  howmany := continent_population(continent_population.LAST);
END;
/
```

Associative arrays help you represent data sets of arbitrary size, with fast lookup for an individual element without knowing its position within the array and without having to loop through all the array elements. It is like a simple version of a SQL table

where you can retrieve values based on the primary key. For simple temporary storage of lookup data, associative arrays let you avoid using the disk space and network operations required for SQL tables.

Because associative arrays are intended for temporary data rather than storing persistent data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`. You can make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body.

How Globalization Settings Affect VARCHAR2 Keys for Associative Arrays

If settings for national language or globalization change during a session that uses associative arrays with `VARCHAR2` key values, the program might encounter a runtime error. For example, changing the `NLS_COMP` or `NLS_SORT` initialization parameters within a session might cause methods such as `NEXT` and `PRIOR` to raise exceptions. If you need to change these settings during the session, make sure to set them back to their original values before performing further operations with these kinds of associative arrays.

When you declare an associative array using a string as the key, the declaration must use a `VARCHAR2`, `STRING`, or `LONG` type. You can use a different type, such as `NCHAR` or `NVARCHAR2`, as the key value to reference an associative array. You can even use a type such as `DATE`, as long as it can be converted to `VARCHAR2` by the `TO_CHAR` function.

However, you must be careful when using other types that the values used as keys are consistent and unique. For example, the string value of `SYSDATE` might change if the `NLS_DATE_FORMAT` initialization parameter changes, so that `array_element(SYSDATE)` does not produce the same result as before. Two different `NVARCHAR2` values might turn into the same `VARCHAR2` value (containing question marks instead of certain national characters). In that case, `array_element(national_string1)` and `array_element(national_string2)` might refer to the same element. Two different `CHAR` or `VARCHAR2` values that differ in terms of case, accented characters, or punctuation characters might also be considered the same if the value of the `NLS_SORT` initialization parameter ends in `_CI` (case-insensitive comparisons) or `_AI` (accent- and case-insensitive comparisons).

When you pass an associative array as a parameter to a remote database using a database link, the two databases can have different globalization settings. When the remote database performs operations such as `FIRST` and `NEXT`, it uses its own character order even if that is different from the order where the collection originated. If character set differences mean that two keys that were unique are not unique on the remote database, the program receives a `VALUE_ERROR` exception.

Choosing Which PL/SQL Collection Types to Use

If you already have code or business logic that uses some other language, you can usually translate that language's array and set types directly to PL/SQL collection types.

- Arrays in other languages become varrays in PL/SQL.
- Sets and bags in other languages become nested tables in PL/SQL.
- Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

When you are writing original code or designing the business logic from the start, you should consider the strengths of each collection type to decide which is appropriate for each situation.

Choosing Between Nested Tables and Associative Arrays

Both nested tables and associative arrays (formerly known as index-by tables) use similar subscript notation, but they have different characteristics when it comes to persistence and ease of parameter passing.

Nested tables can be stored in a database column, but associative arrays cannot. Nested tables can simplify SQL operations where you would normally join a single-column table with a larger table.

Associative arrays are appropriate for relatively small lookup tables where the collection can be constructed in memory each time a procedure is called or a package is initialized. They are good for collecting information whose volume is unknown beforehand, because there is no fixed limit on their size. Their index values are more flexible, because associative array subscripts can be negative, can be nonsequential, and can use string values instead of numbers.

PL/SQL automatically converts between host arrays and associative arrays that use numeric key values. The most efficient way to pass collections to and from the database server is to set up data values in associative arrays, then use those associative arrays with bulk constructs (the FORALL statement or BULK COLLECT clause).

Choosing Between Nested Tables and Varrays

Varrays are a good choice when:

- The number of elements is known in advance.
- The elements are usually all accessed in sequence.

When stored in the database, varrays keep their ordering and subscripts.

Each varray is stored as a single object, either inside the table of which it is a column (if the varray is less than 4KB) or outside the table but still in the same tablespace (if the varray is greater than 4KB). You must update or retrieve all elements of the varray at the same time, which is most appropriate when performing some operation on all the elements at once. But you might find it impractical to store and retrieve large numbers of elements this way.

Nested tables are a good choice when:

- The index values are not consecutive.
- There is no predefined upper bound for index values.
- You need to delete or update some elements, but not all the elements at once.
- You would usually create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

Nested tables can be sparse: you can delete arbitrary elements, rather than just removing an item from the end.

Nested table data is stored in a separate **store table**, a system-generated database table associated with the nested table. The database joins the tables for you when you access the nested table. This makes nested tables suitable for queries and updates that only affect some elements of the collection.

You cannot rely on the order and subscripts of a nested table remaining stable as the nested table is stored in and retrieved from the database, because the order and subscripts are not preserved in the database.

Defining Collection Types

To create collections, you define a collection type, then declare variables of that type. You can define TABLE and VARRAY types in the declarative part of any PL/SQL block, subprogram, or package.

Collections follow the same scoping and instantiation rules as other types and variables. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

Nested Tables

To define a PL/SQL type for nested tables, use the syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
```

type_name is a type specifier used later to declare collections. For nested tables declared within PL/SQL, *element_type* is any PL/SQL datatype except:

```
REF CURSOR
```

Nested tables declared in SQL using the CREATE TYPE statement have additional restrictions on the element type. They cannot use the following element types:

```
BINARY_INTEGER, PLS_INTEGER
BOOLEAN
LONG, LONG RAW
NATURAL, NATURALN
POSITIVE, POSITIVEN
REF CURSOR
SIGNTYPE
STRING
```

Varrays

To define a PL/SQL type for varrays, use the syntax:

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit)
   OF element_type [NOT NULL];
```

The meanings of *type_name* and *element_type* are the same as for nested tables.

size_limit is a positive integer literal representing the maximum number of elements in the array.

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;
BEGIN
   NULL;
END;
/
```

Associative Arrays

Associative arrays (also known as index-by tables) let you insert elements using arbitrary key values. The keys do not have to be consecutive. They use the syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
   INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(size_limit)];
   INDEX BY key_type;
```

The key_type can be numeric, either PLS_INTEGER or BINARY_INTEGER. It can also be VARCHAR2 or one of its subtypes VARCHAR, STRING, or LONG. You must specify the length of a VARCHAR2-based key, except for LONG which is equivalent to declaring a key type of VARCHAR2(32760). The types RAW, LONG RAW, ROWID, CHAR, and CHARACTER are not allowed as keys for an associative array.

An initialization clause is not allowed. There is no constructor notation for associative arrays.

When you reference an element of an associative array that uses a VARCHAR2-based key, you can use other types, such as DATE or TIMESTAMP, as long as they can be converted to VARCHAR2 with the TO_CHAR function.

Associative arrays can store data using a primary key value as the index, where the key values are not sequential. The example below creates a single element in an associative array, with a subscript of 100 rather than 1:

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE
      INDEX BY PLS_INTEGER;
   emp_tab EmpTabTyp;
BEGIN
   /* Retrieve employee record. */
   SELECT * INTO emp_tab(100) FROM employees WHERE employee_id = 100;
END;
/
```

Defining SQL Types Equivalent to PL/SQL Collection Types

To store nested tables and varrays inside database tables, you must also declare SQL types using the CREATE TYPE statement. The SQL types can be used as columns or as attributes of SQL object types.

You can declare equivalent types within PL/SQL, or use the SQL type name in a PL/SQL variable declaration.

Example 5–1 Declaring a Nested Table in SQL

The following SQL*Plus script shows how you might declare a nested table in SQL, and use it as an attribute of an object type:

```
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
/

DROP TYPE Student;
```

```
DROP TYPE CourseList;
```

The identifier `courses` represents an entire nested table. Each element of `courses` stores the name of a college course such as `'Math 1020'`.

Example 5–2 Creating a Table with a Varray Column

The script below creates a database column that stores varrays. Each varray element contains a VARCHAR2.

```
-- Each project has a 16-character code name.
-- We will store up to 50 projects at a time in a database column.
CREATE TYPE ProjectList AS VARRAY(50) OF VARCHAR2(16);
/
CREATE TABLE department (  -- create database table
   dept_id  NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
-- Each department can have up to 50 projects.
   projects ProjectList);

DROP TABLE department;
DROP TYPE ProjectList;
```

Declaring PL/SQL Collection Variables

After defining a collection type, you declare variables of that type. You use the new type name in the declaration, the same as with predefined types such as NUMBER.

Example 5–3 Declaring Nested Tables, Varrays, and Associative Arrays

```
DECLARE
   TYPE nested_type IS TABLE OF VARCHAR2(20);
   TYPE varray_type IS VARRAY(5) OF INTEGER;
   TYPE assoc_array_num_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   TYPE assoc_array_str_type IS TABLE OF VARCHAR2(32) INDEX BY PLS_INTEGER;
   TYPE assoc_array_str_type2 IS TABLE OF VARCHAR2(32) INDEX BY VARCHAR2(64);
   v1 nested_type;
   v2 varray_type;
   v3 assoc_array_num_type;
   v4 assoc_array_str_type;
   v5 assoc_array_str_type2;
BEGIN
   v1 := nested_type('Arbitrary','number','of','strings');
   v2 := varray_type(10, 20, 40, 80, 160); -- Up to 5 integers
   v3(99) := 10; -- Just start assigning to elements.
   v3(7) := 100; -- Subscripts can be any integer values.
   v4(42) := 'Cat'; -- Just start assigning to elements.
   v4(54) := 'Hat'; -- Subscripts can be any integer values.
   v5('Canada') := 'North America'; -- Just start assigning to elements.
   v5('Greece') := 'Europe';        -- Subscripts can be string values.
END;
/
```

Example 5–4 Declaring Collections with %TYPE

You can use %TYPE to specify the datatype of a previously declared collection, so that changing the definition of the collection automatically updates other variables that depend on the number of elements or the element type:

```
DECLARE
   TYPE Few_Colors IS VARRAY(10) OF VARCHAR2(20);
   TYPE Many_Colors IS VARRAY(100) OF VARCHAR2(64);
   some_colors Few_Colors;
-- If we change the type of SOME_COLORS from FEW_COLORS to MANY_COLORS,
-- RAINBOW and CRAYONS will use the same type when this block is recompiled.
   rainbow some_colors%TYPE;
   crayons some_colors%TYPE;
BEGIN
   NULL;
END;
/
```

Example 5–5 Declaring a Procedure Parameter as a Nested Table

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. The following example declares a nested table as a parameter of a packaged procedure:

```
CREATE PACKAGE personnel AS
   TYPE Staff_List IS TABLE OF employees.employee_id%TYPE;
   PROCEDURE award_bonuses (who_gets_em IN Staff_List);
END personnel;
/

DROP PACKAGE personnel;
```

To call PERSONNEL.AWARD_BONUSES from outside the package, you declare a variable of type PERSONNEL.STAFF and pass that variable as the parameter.

You can also specify a collection type in the RETURN clause of a function specification.

Example 5–6 Specifying Collection Element Types with %TYPE and %ROWTYPE

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
-- Nested table type that can hold an arbitrary number of
-- employee IDs. The element type is based on a column from
-- the EMPLOYEES table. We do not need to know whether the
-- ID is a number or a string.
   TYPE EmpList IS TABLE OF employees.employee_id%TYPE;

-- Array type that can hold information about 10 employees.
-- The element type is a record that contains all the same
-- fields as the EMPLOYEES table.
   TYPE Top_Salespeople IS VARRAY(10) OF employees%ROWTYPE;

-- Declare a cursor to select a subset of columns.
   CURSOR c1 IS SELECT first_name, last_name FROM employees;
-- Array type that can hold a list of names. The element type
-- is a record that contains the same fields as the cursor
```

```
-- (that is, first_name and last_name).
   TYPE NameList IS VARRAY(20) OF c1%ROWTYPE;
BEGIN
   NULL;
END;
/
```

Example 5–7 VARRAY of Records

This example uses a RECORD type to specify the element type:

```
DECLARE
   TYPE GlossEntry IS RECORD ( term VARCHAR2(20), meaning VARCHAR2(200) );
   TYPE Glossary IS VARRAY(250) OF GlossEntry;
BEGIN
   NULL;
END;
/
```

Example 5–8 NOT NULL Constraint on Collection Elements

You can also impose a NOT NULL constraint on the element type:

```
DECLARE
   TYPE EmpList IS TABLE OF employees.employee_id%TYPE NOT NULL;
   my_employees EmpList := EmpList(100, 150, 160, 200);
BEGIN
   my_employees(3) := NULL; -- Assigning NULL raises an exception
END;
/
```

Initializing and Referencing Collections

Until you initialize it, a nested table or varray is atomically null: the collection itself is null, not its elements. To initialize a nested table or varray, you use a *constructor*, a system-defined function with the same name as the collection type. This function "constructs" collections from the elements passed to it.

You must explicitly call a constructor for each varray and nested table variable. (Associative arrays, the third kind of collection, do not use constructors.) Constructor calls are allowed wherever function calls are allowed.

Example 5–9 Constructor for a Nested Table

The following example initializes a nested table using a constructor, which looks like a function with the same name as the collection type:

```
DECLARE
   TYPE Colors IS TABLE OF VARCHAR2(16);
   rainbow Colors;
BEGIN
   rainbow := Colors('Red','Orange','Yellow','Green','Blue','Indigo','Violet');
END;
/
```

Because a nested table does not have a declared maximum size, you can put as many elements in the constructor as necessary.

Example 5–10 Constructor for a Varray

This example initializes a varray using a constructor, which looks like a function with the same name as the collection type:

```
DECLARE
-- In the varray, we put an upper limit on the number of elements.
   TYPE Colors IS VARRAY(10) OF VARCHAR2(16);
   rainbow Colors;
BEGIN
-- Since COLORS is declared as VARRAY(10), we can put up to 10
-- elements in the constructor.
   rainbow := Colors('Red','Orange','Yellow','Green','Blue','Indigo','Violet');
END;
/
```

Example 5–11 Collection Constructor Including Null Elements

Unless you impose the NOT NULL constraint in the type declaration, you can pass null elements to a constructor:

```
DECLARE
   TYPE Colors IS TABLE OF VARCHAR2(20);
   my_colors Colors;
   TYPE ColorsNoNulls IS TABLE OF VARCHAR2(20) NOT NULL;
BEGIN
   my_colors := Colors('Sienna',NULL,'Teal','Umber',NULL);
-- If MY_COLORS was of type ColorsNoNulls, we could not include
-- null values in the constructor.
END;
/
```

Example 5–12 Combining Collection Declaration and Constructor

You can initialize a collection in its declaration, which is a good programming practice:

```
DECLARE
   TYPE Colors IS TABLE OF VARCHAR2(20);
   my_colors Colors := Colors('Brown','Gray','Beige');
BEGIN
   NULL;
END;
/
```

Example 5–13 Empty Varray Constructor

If you call a constructor without arguments, you get an empty but non-null collection:

```
DECLARE
   TYPE Colors IS VARRAY(100) OF VARCHAR2(20);
   my_colors Colors;
BEGIN
   IF my_colors IS NULL THEN
      dbms_output.put_line('Before initialization, the varray is null.');
-- While the varray is null, we can't check its COUNT attribute.
--      dbms_output.put_line('It has ' || my_colors.COUNT || ' elements.');
   ELSE
      dbms_output.put_line('Before initialization, the varray is not null.');
   END IF;
   my_colors := Colors(); -- initialize empty varray
```

```
   IF my_colors IS NULL THEN
      dbms_output.put_line('After initialization, the varray is null.');
   ELSE
      dbms_output.put_line('After initialization, the varray is not null.');
      dbms_output.put_line('It has ' || my_colors.COUNT || ' elements.');
   END IF;
END;
/
```

In this case, you can call the collection's EXTEND method to add elements later.

Example 5–14 Nested Table Constructor Within a SQL Statement

In this example, you insert several scalar values and a `CourseList` nested table into the SOPHOMORES table.

```
BEGIN
   INSERT INTO sophomores
      VALUES (5035, 'Janet Alvarez', '122 Broad St', 'FT',
         CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100'));
```

Example 5–15 Varray Constructor Within a SQL Statement

In this example, you insert a row into database table DEPARTMENT. The varray constructor `ProjectList()` provides a value for column PROJECTS.

```
BEGIN
  INSERT INTO department
    VALUES(60, 'Security', 750400,
      ProjectList('New Badges', 'Track Computers', 'Check Exits'));
```

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 in most cases, or a VARCHAR2 for associative arrays declared with strings as keys.

The allowed subscript ranges are:

- For nested tables, 1 .. 2**31.
- For varrays, 1 .. *size_limit*, where you specify the limit in the declaration.
- For associative arrays with a numeric key, -2**31 .. 2**31.
- For associative arrays with a string key, the length of the key and number of possible values depends on the VARCHAR2 length limit in the type declaration, and the database character set.

Example 5–16 Referencing a Nested Table Element By Subscript

This example shows how to reference an element in the nested table NAMES:

```
DECLARE
   TYPE Roster IS TABLE OF VARCHAR2(15);
   names Roster := Roster('J Hamil', 'D Caruso', 'R Singh');
```

```
BEGIN
  FOR i IN names.FIRST .. names.LAST
  LOOP
      IF names(i) = 'J Hamil' THEN
        NULL;
      END IF;
  END LOOP;
END;
/
```

Example 5–17 Passing a Nested Table Element as a Parameter

This 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 := 2;
BEGIN
   verify_name(names(i));  -- call procedure
END;
/
```

Assigning Collections

One collection can be assigned to another by an `INSERT`, `UPDATE`, `FETCH`, or `SELECT` statement, an assignment statement, or a subprogram call.

You can assign the value of an expression to a specific element in a collection using the syntax:

collection_name(*subscript*) := *expression*;

where *expression* yields a value of the type specified for elements in the collection type definition.

You can use operators such as `SET`, `MULTISET UNION`, `MULTISET INTERSECT`, and `MULTISET EXCEPT` to transform nested tables as part of an assignment statement.

Example 5–18 Datatype Compatibility for Collection Assignment

This example shows that collections must have the same datatype for an assignment to work. Having the same element type is not enough.

```
DECLARE
   TYPE last_name_typ IS VARRAY(3) OF VARCHAR2(64);
   TYPE surname_typ IS VARRAY(3) OF VARCHAR2(64);
-- These first two variables have the same datatype.
   group1 last_name_typ := last_name_typ('Jones','Wong','Marceau');
   group2 last_name_typ := last_name_typ('Klein','Patsos','Singh');
-- This third variable has a similar declaration, but is not the same type.
   group3 surname_typ := surname_typ('Trevisi','Macleod','Marquez');
BEGIN
-- Allowed because they have the same datatype
   group1 := group2;
-- Not allowed because they have different datatypes
--   group3 := group2;
END;
```

```
/
```

Example 5–19 Assigning a Null Value to a Nested Table

If you assign an atomically null nested table or varray to a second nested table or varray, the second collection must be reinitialized:

```
DECLARE
   TYPE Colors IS TABLE OF VARCHAR2(64);
-- This nested table has some values.
   crayons Colors := Colors('Silver','Gold');
-- This nested table is not initialized ("atomically null").
   empty_set Colors;
BEGIN
-- At first, the initialized variable is not null.
   if crayons IS NOT NULL THEN
      dbms_output.put_line('OK, at first crayons is not null.');
   END IF;

-- Then we assign a null nested table to it.
   crayons := empty_set;

-- Now it is null.
   if crayons IS NULL THEN
      dbms_output.put_line('OK, now crayons has become null.');
   END IF;

-- We must use another constructor to give it some values.
   crayons := Colors('Yellow','Green','Blue');
END;
/
```

In the same way, assigning the value `NULL` to a collection makes it atomically null.

Example 5–20 Possible Exceptions for Collection Assignments

Assigning a value to a collection element can cause various exceptions:

- If the subscript is null or is not convertible to the right datatype, PL/SQL raises the predefined exception `VALUE_ERROR`. Usually, the subscript must be an integer. Associative arrays can also be declared to have `VARCHAR2` subscripts.
- If the subscript refers to an uninitialized element, PL/SQL raises `SUBSCRIPT_BEYOND_COUNT`.
- If the collection is atomically null, PL/SQL raises `COLLECTION_IS_NULL`.

```
DECLARE
   TYPE WordList IS TABLE OF VARCHAR2(5);
   words WordList;
BEGIN
   /* Assume execution continues despite the raised exceptions. */
-- Raises COLLECTION_IS_NULL. We haven't used a constructor yet.
-- This exception applies to varrays and nested tables, but not to
-- associative arrays which don't need a constructor.
   words(1) := 10;
-- After using a constructor, we can assign values to the elements.
   words := WordList(10,20,30);
-- Any expression that returns a VARCHAR2(5) is OK.
   words(1) := 'yes';
   words(2) := words(1) || 'no';
```

```
-- Raises VALUE_ERROR because the assigned value is too long.
   words(3) := 'longer than 5 characters';
-- Raises VALUE_ERROR because the subscript of a nested table must
-- be an integer.
   words('B') := 'dunno';
-- Raises SUBSCRIPT_BEYOND_COUNT because we only made 3 elements
-- in the constructor. To add new ones, we must call the EXTEND
-- method first.
   words(4) := 'maybe';
END;
/
```

Example 5–21 Assigning Nested Tables with Set Operators

This example shows some of the ANSI-standard operators that you can apply to nested tables:

```
DECLARE
  TYPE nested_typ IS TABLE OF NUMBER;
  nt1 nested_typ := nested_typ(1,2,3);
  nt2 nested_typ := nested_typ(3,2,1);
  nt3 nested_typ := nested_typ(2,3,1,3);
  nt4 nested_typ := nested_typ(1,2,4);
  answer nested_typ;

-- The results might be in a different order than you expect.
-- (Remember, you should not rely on the order of elements in nested tables.)
  PROCEDURE print_nested_table(the_nt nested_typ) IS
     output VARCHAR2(128);
  BEGIN
     IF the_nt IS NULL THEN
        dbms_output.put_line('Results: <NULL>');
        RETURN;
     END IF;
     IF the_nt.COUNT = 0 THEN
        dbms_output.put_line('Results: empty set');
        RETURN;
     END IF;
     FOR i IN the_nt.FIRST .. the_nt.LAST
     LOOP
        output := output || the_nt(i) || ' ';
     END LOOP;
     dbms_output.put_line('Results: ' || output);
  END;
BEGIN
  answer := nt1 MULTISET UNION nt4; -- (1,2,3,1,2,4)
  print_nested_table(answer);
  answer := nt1 MULTISET UNION nt3; -- (1,2,3,2,3,1,3)
  print_nested_table(answer);
  answer := nt1 MULTISET UNION DISTINCT nt3; -- (1,2,3)
  print_nested_table(answer);

  answer := nt2 MULTISET INTERSECT nt3; -- (3,2,1)
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT DISTINCT nt3; -- (3,2,1)
  print_nested_table(answer);

  answer := SET(nt3); -- (2,3,1)
  print_nested_table(answer);
```

```
  answer := nt3 MULTISET EXCEPT nt2; -- (3)
  print_nested_table(answer);
  answer := nt3 MULTISET EXCEPT DISTINCT nt2; -- ()
  print_nested_table(answer);

END;
/
```

Comparing Collections

You can check whether a collection is null, and whether two collections are the same. Comparisons such as greater than, less than, and so on are not allowed.

This restriction also applies to implicit comparisons. For example, collections cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` list.

If you want to do such comparison operations, you must define your own notion of what it means for collections to be greater than, less than, and so on, and write one or more functions to examine the collections and their elements and return a true or false value.

You can apply set operators (`CARDINALITY`, `MEMBER OF`, `IS A SET`, `IS EMPTY`) to check certain conditions within a nested table or between two nested tables.

Example 5–22 Checking if a Collection Is Null

Nested tables and varrays can be atomically null, so they can be tested for nullity:

```
DECLARE
   TYPE Staff IS TABLE OF Employee;
   members Staff;
BEGIN
  -- Condition yields TRUE because we haven't used a constructor.
   IF members IS NULL THEN ...
END;
```

Example 5–23 Comparing Two Collections

Collections can be compared for equality or inequality. They cannot be ordered, because there is no "greater than" or "less than" comparison.

```
DECLARE
   TYPE Colors IS TABLE OF VARCHAR2(64);
   primaries Colors := Colors('Blue','Green','Red');
   rgb Colors := Colors('Red','Green','Blue');
   traffic_light Colors := Colors('Red','Green','Amber');
BEGIN
-- We can use = or !=, but not < or >.
-- Notice that these 2 are equal even though the members are in different order.
   IF primaries = rgb THEN
      dbms_output.put_line('OK, PRIMARIES and RGB have the same members.');
   END IF;
   IF rgb != traffic_light THEN
      dbms_output.put_line('OK, RGB and TRAFFIC_LIGHT have different members.');
   END IF;
END;
/
```

Example 5–24 Comparing Nested Tables with Set Operators

You can test certain properties of a nested table, or compare two nested tables, using ANSI-standard set operations:

```
DECLARE
  TYPE nested_typ IS TABLE OF NUMBER;
  nt1 nested_typ := nested_typ(1,2,3);
  nt2 nested_typ := nested_typ(3,2,1);
  nt3 nested_typ := nested_typ(2,3,1,3);
  nt4 nested_typ := nested_typ(1,2,4);
  answer BOOLEAN;
  howmany NUMBER;
  PROCEDURE testify(truth BOOLEAN DEFAULT NULL, quantity NUMBER DEFAULT NULL) IS
  BEGIN
     IF truth IS NOT NULL THEN
        dbms_output.put_line(CASE truth WHEN TRUE THEN 'True' WHEN FALSE THEN
'False' END);
     END IF;
     IF quantity IS NOT NULL THEN
        dbms_output.put_line(quantity);
     END IF;
  END;
BEGIN
  answer := nt1 IN (nt2,nt3,nt4); -- true, nt1 matches nt2
  testify(truth => answer);
  answer := nt1 SUBMULTISET OF nt3; -- true, all elements match
  testify(truth => answer);
  answer := nt1 NOT SUBMULTISET OF nt4; -- also true
  testify(truth => answer);

  howmany := CARDINALITY(nt3); -- number of elements in nt3
  testify(quantity => howmany);
  howmany := CARDINALITY(SET(nt3)); -- number of distinct elements
  testify(quantity => howmany);

  answer := 4 MEMBER OF nt1; -- false, no element matches
  testify(truth => answer);
  answer := nt3 IS A SET; -- false, nt3 has duplicates
  testify(truth => answer);
  answer := nt3 IS NOT A SET; -- true, nt3 has duplicates
  testify(truth => answer);
  answer := nt1 IS EMPTY; -- false, nt1 has some members
  testify(truth => answer);
END;
/
```

Using PL/SQL Collections with SQL Statements

Collections let you manipulate complex datatypes within PL/SQL. Your program can compute subscripts to process specific elements in memory, and use SQL to store the results in database tables.

Example 5–25 Creating a SQL Type Corresponding to a PL/SQL Nested Table

In SQL*Plus, you can create SQL types whose definitions correspond to PL/SQL nested tables and varrays:

```
SQL> CREATE TYPE CourseList AS TABLE OF VARCHAR2(64);
```

You can use these SQL types as columns in database tables:

```
SQL> CREATE TABLE department (
  2  name     VARCHAR2(20),
  3  director VARCHAR2(20),
  4  office   VARCHAR2(20),
  5  courses  CourseList)
  6  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 whenever a database table has a nested table column. The clause identifies the nested table and names a system-generated store table, in which Oracle stores the nested table data.

Example 5–26 Inserting a Nested Table into a Database Table

Now, you can populate the database table. The table constructor provides values that all go into the single column COURSES:

```
BEGIN
   INSERT INTO department
      VALUES('English', 'Lynn Saunders', 'Breakstone Hall 205',
         CourseList('Expository Writing',
                    'Film and Literature',
                    'Modern Science Fiction',
                    'Discursive Writing',
                    'Modern English Grammar',
                    'Introduction to Shakespeare',
                    'Modern Drama',
                    'The Short Story',
                    'The American Novel'));
END;
```

Example 5–27 Using PL/SQL Nested Tables with INSERT, UPDATE, DELETE, and SELECT Statements

You can retrieve all the courses offered by the English department into a PL/SQL nested table:

```
CREATE TYPE ColorList AS TABLE OF VARCHAR2(64);
/
CREATE TABLE flowers (name VARCHAR2(20), colors ColorList) NESTED TABLE colors
STORE AS colors_tab;

BEGIN
   INSERT INTO flowers VALUES('Rose', ColorList('Red','Yellow','White'));
   INSERT INTO flowers VALUES('Tulip', ColorList('Red','White','Yellow', 'Blue'));
   INSERT INTO flowers VALUES('Iris', ColorList('White','Purple'));
   COMMIT;
END;
/

DECLARE
-- This type declaration is not needed, because PL/SQL can see the SQL type.
--   TYPE ColorList IS TABLE OF VARCHAR2(64);
-- Declare a variable that can hold a set of colors.
   my_colors ColorList;
-- Declare a record that can hold a row from the table.
-- One of the record fields is a set of colors.
   my_flower flowers%ROWTYPE;
```

```
   new_colors ColorList;
BEGIN
-- Look up a name and query just the associated colors.
   SELECT colors INTO my_colors FROM flowers WHERE name = 'Rose';
   FOR i IN my_colors.FIRST .. my_colors.LAST
   LOOP
      dbms_output.put_line('Rose color = ' || my_colors(i));
   END LOOP;

-- Look up a name and query the entire row.
   SELECT * INTO my_flower FROM flowers WHERE name = 'Iris';
-- Now COLORS is a field in a record, so we access it with dot notation.
   FOR i IN my_flower.colors.FIRST .. my_flower.colors.LAST
   LOOP
-- Because we have all the table columns in the record, we can refer to NAME also.
      dbms_output.put_line(my_flower.name || ' color = ' || my_flower.colors(i));
   END LOOP;

-- We can replace a set of colors by making a new collection and using it
-- in an UPDATE statement.
   new_colors := ColorList('Red','Yellow','White','Pink');
   UPDATE flowers SET colors = new_colors WHERE name = 'Rose';

-- Or we can modify the original collection and use it in the UPDATE.
-- We'll add a new final element and fill in a value.
   my_flower.colors.EXTEND(1);
   my_flower.colors(my_flower.colors.COUNT) := 'Yellow';
   UPDATE flowers SET colors = my_flower.colors WHERE name = my_flower.name;

-- We can even treat the nested table column like a real table and
-- insert, update, or delete elements.
-- The TABLE operator makes the statement apply to the nested table produced by
the subquery.

   INSERT INTO TABLE(SELECT colors FROM flowers WHERE name = 'Rose')
VALUES('Black');
   DELETE FROM TABLE(SELECT colors FROM flowers WHERE name = 'Rose') WHERE column_
value = 'Yellow';
   UPDATE TABLE(SELECT colors FROM flowers WHERE name = 'Iris')
      SET column_value = 'Indigo' WHERE column_value = 'Purple';

   COMMIT;
END;
/

DROP TABLE flowers;
DROP TYPE ColorList;
```

Within PL/SQL, you can manipulate the nested table by looping through its elements, using methods such as `TRIM` or `EXTEND`, and updating some or all of the elements. Afterwards, you can store the updated table in the database again.

Example 5–28 Updating a Nested Table within a Database Table

You can revise the list of courses offered by the English Department:

```
DECLARE
   new_courses CourseList :=
      CourseList('Expository Writing',
                 'Film and Literature',
                 'Discursive Writing',
```

```
                'Modern English Grammar',
                'Realism and Naturalism',
                'Introduction to Shakespeare',
                'Modern Drama',
                'The Short Story',
                'The American Novel',
                '20th-Century Poetry',
                'Advanced Workshop in Poetry');
BEGIN
   UPDATE department
      SET courses = new_courses WHERE name = 'English';
END;
```

Using PL/SQL Varrays with INSERT, UPDATE, and SELECT Statements

This example shows how you can transfer varrays between PL/SQL variables and SQL tables. You can insert table rows containing varrays, update a row to replace its varray, and select varrays into PL/SQL variables. You cannot update or delete individual varray elements directly with SQL; you have to select the varray from the table, change it in PL/SQL, then update the table to include the new varray.

```
-- By using a varray, we put an upper limit on the number of elements
-- and ensure they always come back in the same order.
CREATE TYPE RainbowTyp AS VARRAY(7) OF VARCHAR2(64);
/

CREATE TABLE rainbows (language VARCHAR2(64), colors RainbowTyp);

BEGIN
   INSERT INTO rainbows VALUES('English',
RainbowTyp('Red','Orange','Yellow','Green','Blue','Indigo','Violet'));
   INSERT INTO rainbows VALUES('Francais',
RainbowTyp('Rouge','Orange','Jaune','Vert','Bleu','Indigo','Violet'));
   COMMIT;
END;
/

DECLARE
   new_colors RainbowTyp :=
RainbowTyp('Crimson','Orange','Amber','Forest','Azure','Indigo','Violet');
   some_colors RainbowTyp;
BEGIN
   UPDATE rainbows SET colors  = new_colors WHERE language = 'English';
   COMMIT;
   SELECT colors INTO some_colors FROM rainbows WHERE language = 'Francais';
   FOR i IN some_colors.FIRST .. some_colors.LAST
   LOOP
      dbms_output.put_line('Color = ' || some_colors(i));
   END LOOP;
END;
/

DROP TABLE rainbows;
DROP TYPE RainbowTyp;
```

Manipulating Individual Collection Elements with SQL

By default, SQL operations store and retrieve whole collections rather than individual elements. To manipulate the individual elements of a collection with SQL, use the TABLE operator. The TABLE operator uses a subquery to extract the varray or nested table, so that the INSERT, UPDATE, or DELETE statement applies to the nested table rather than the top-level table.

Example 5–29 Performing INSERT, UPDATE, and DELETE Operations on PL/SQL Nested Tables

To perform DML operations on a PL/SQL nested table, use the operators TABLE and CAST. This way, you can do set operations on nested tables using SQL notation, without actually storing the nested tables in the database.

The operands of CAST are PL/SQL collection variable and a SQL collection type (created by the CREATE TYPE statement). CAST converts the PL/SQL collection to the SQL type.

```
DECLARE
   revised CourseList :=
      CourseList(Course(1002, 'Expository Writing', 3),
                 Course(2020, 'Film and Literature', 4),
                 Course(4210, '20th-Century Poetry', 4),
                 Course(4725, 'Advanced Workshop in Poetry', 5));
   num_changed INTEGER;
BEGIN
   SELECT COUNT(*) INTO num_changed
      FROM TABLE(CAST(revised AS CourseList)) new,
      TABLE(SELECT courses FROM department
         WHERE name = 'English') AS old
      WHERE new.course_no = old.course_no AND
         (new.title != old.title OR new.credits != old.credits);
   dbms_output.put_line(num_changed);
END;
```

Using Multilevel Collections

In addition to collections of scalar or object types, you can also create collections whose elements are collections. For example, you can create a nested table of varrays, a varray of varrays, a varray of nested tables, and so on.

When creating a nested table of nested tables as a column in SQL, check the syntax of the CREATE TABLE statement to see how to define the storage table.

Here are some examples showing the syntax and possibilities for multilevel collections.

Example 5–30 Multilevel VARRAY

```
declare
 type t1 is varray(10) of integer;
 type nt1 is varray(10) of t1; -- multilevel varray type
 va t1 := t1(2,3,5);
-- initialize multilevel varray
 nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
 i integer;
 va1 t1;
begin
 -- multilevel access
```

```
 i := nva(2)(3); -- i will get value 73
 dbms_output.put_line('I = ' || i);
 -- add a new varray element to nva
 nva.extend;

-- replace inner varray elements
 nva(5) := t1(56, 32);
 nva(4) := t1(45,43,67,43345);
-- replace an inner integer element
 nva(4)(4) := 1; -- replaces 43345 with 1
-- add a new element to the 4th varray element
-- and store integer 89 into it.
 nva(4).extend;
 nva(4)(5) := 89;
end;
/
```

Example 5–31 Multilevel Nested Table

```
declare
 type tb1 is table of varchar2(20);
 type ntb1 is table of tb1; -- table of table elements
 type tv1 is varray(10) of integer;
 type ntb2 is table of tv1; -- table of varray elements

 vtb1 tb1 := tb1('one', 'three');
 vntb1 ntb1 := ntb1(vtb1);
 vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3));  -- table of varray elements
begin
 vntb1.extend;
 vntb1(2) := vntb1(1);
-- delete the first element in vntb1
 vntb1.delete(1);
-- delete the first string from the second table in the nested table
 vntb1(2).delete(1);
end;
/
```

Example 5–32 Multilevel Associative Array

```
declare
 type tb1 is table of integer index by binary_integer;
-- the following is index-by table of index-by tables
 type ntb1 is table of tb1 index by binary_integer;
 type va1 is varray(10) of varchar2(20);
-- the following is index-by table of varray elements
 type ntb2 is table of va1 index by binary_integer;

 v1 va1 := va1('hello', 'world');
 v2 ntb1;
 v3 ntb2;
 v4 tb1;
 v5 tb1; -- empty table
begin
 v4(1) := 34;
 v4(2) := 46456;
 v4(456) := 343;
 v2(23) := v4;
 v3(34) := va1(33, 456, 656, 343);
```

```
-- assign an empty table to v2(35) and try again
   v2(35) := v5;
   v2(35)(2) := 78; -- it works now
end;
/
```

Example 5–33 Multilevel Collections and Bulk 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;
/

drop table tab1;
drop type t1;
```

Using Collection Methods

These collection methods make collections easier to use, and make your applications easier to maintain:

EXISTS
COUNT
LIMIT
FIRST and LAST
PRIOR and NEXT
EXTEND
TRIM
DELETE

A **collection method** is a built-in function or procedure that operates on collections and is called using dot notation.

Collection methods cannot be called from SQL statements.

`EXTEND` and `TRIM` cannot be used with associative arrays.

`EXISTS`, `COUNT`, `LIMIT`, `FIRST`, `LAST`, `PRIOR`, and `NEXT` are functions; `EXTEND`, `TRIM`, and `DELETE` are procedures.

`EXISTS`, `PRIOR`, `NEXT`, `TRIM`, `EXTEND`, and `DELETE` take parameters corresponding to collection subscripts, which are usually integers but can also be strings for associative arrays.

Only `EXISTS` can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises `COLLECTION_IS_NULL`.

Checking If a Collection Element Exists (EXISTS Method)

EXISTS(n) returns TRUE if the *n*th element in a collection exists. Otherwise, EXISTS(n) returns FALSE. By combining EXISTS with DELETE, you can work with sparse nested tables. You can also use EXISTS to avoid referencing a nonexistent element, which raises an exception. When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

```
DECLARE
   TYPE NumList IS TABLE OF INTEGER;
   n NumList := NumList(1,3,5,7);
BEGIN
   n.DELETE(2); -- Delete the second element
   IF n.EXISTS(1) THEN
      dbms_output.put_line('OK, element #1 exists.');
   END IF;
   IF n.EXISTS(2) = FALSE THEN
      dbms_output.put_line('OK, element #2 has been deleted.');
   END IF;
   IF n.EXISTS(99) = FALSE THEN
      dbms_output.put_line('OK, element #99 does not exist at all.');
   END IF;
END;
/
```

Counting the Elements in a Collection (COUNT Method)

COUNT returns the number of elements that a collection currently contains:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(2,4,6,8); -- Collection starts with 4 elements.
BEGIN
   dbms_output.put_line('There are ' || n.COUNT || ' elements in N.');
   n.EXTEND(3); -- Add 3 new elements at the end.
   dbms_output.put_line('Now there are ' || n.COUNT || ' elements in N.');
   n := NumList(86,99); -- Assign a completely new value with 2 elements.
   dbms_output.put_line('Now there are ' || n.COUNT || ' elements in N.');
   n.TRIM(2); -- Remove the last 2 elements, leaving none.
   dbms_output.put_line('Now there are ' || n.COUNT || ' elements in N.');
END;
/
```

COUNT is useful because the current size of a collection is not always known. For example, you can fetch a column of Oracle data into a nested table, where the number of elements depends on the size of the result set.

For varrays, COUNT always equals LAST. You can increase or decrease the size of a varray using the EXTEND and TRIM methods, so the value of COUNT can change, up to the value of the LIMIT method.

For nested tables, COUNT normally equals LAST. But, if you delete elements from the middle of a nested table, COUNT becomes smaller than LAST. When tallying elements, COUNT ignores deleted elements.

Checking the Maximum Size of a Collection (LIMIT Method)

For nested tables and associative arrays, which have no maximum size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can

contain/ You specify this limit in the type definition, and can change it later with the TRIM and EXTEND methods. For instance, if the maximum size of varray PROJECTS is 25 elements, the following IF condition is true:

```
DECLARE
   TYPE Colors IS VARRAY(7) OF VARCHAR2(64);
   c Colors := Colors('Gold','Silver');
BEGIN
   dbms_output.put_line('C has ' || c.COUNT || ' elements now.');
   dbms_output.put_line('C''s type can hold a maximum of ' || c.LIMIT || '
elements.');
   dbms_output.put_line('The maximum number you can use with C.EXTEND() is ' ||
(c.LIMIT - c.COUNT));
END;
/
```

Finding the First or Last Collection Element (FIRST and LAST Methods)

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection that uses integer subscripts.

For an associative array with VARCHAR2 key values, the lowest and highest key values are returned. By default, the order is based on the binary values of the characters in the string. If the NLS_COMP initialization parameter is set to ANSI, the order is based on the locale-specific sort order specified by the NLS_SORT initialization parameter.

If the collection is empty, FIRST and LAST return NULL.

If the collection contains only one element, FIRST and LAST return the same index value.

The following example shows how to use FIRST and LAST to iterate through the elements in a collection that has consecutive subscripts:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1,3,5,7);
   counter INTEGER;
BEGIN
   dbms_output.put_line('N''s first subscript is ' || n.FIRST);
   dbms_output.put_line('N''s last subscript is ' || n.LAST);

-- When the subscripts are consecutive starting at 1, it's simple to loop through
them.
   FOR i IN n.FIRST .. n.LAST
   LOOP
      dbms_output.put_line('Element #' || i || ' = ' || n(i));
   END LOOP;

   n.DELETE(2); -- Delete second element.
-- When the subscripts have gaps or the collection might be uninitialized,
-- the loop logic is more extensive. We start at the first element, and
-- keep looking for the next element until there are no more.
   IF n IS NOT NULL THEN
      counter := n.FIRST;
      WHILE counter IS NOT NULL
      LOOP
         dbms_output.put_line('Element #' || counter || ' = ' || n(counter));
         counter := n.NEXT(counter);
      END LOOP;
```

```
   ELSE
      dbms_output.put_line('N is null, nothing to do.');
   END IF;
END;
/
```

For varrays, FIRST always returns 1 and LAST always equals COUNT.

For nested tables, normally FIRST returns 1 and LAST equals COUNT. But if you delete elements from the beginning of a nested table, FIRST returns a number larger than 1. If you delete elements from the middle of a nested table, LAST becomes larger than COUNT.

When scanning elements, FIRST and LAST ignore deleted elements.

Looping Through Collection Elements (PRIOR and NEXT Methods)

PRIOR(n) returns the index number that precedes index n in a collection. NEXT(n) returns the index number that succeeds index n. If n has no predecessor, PRIOR(n) returns NULL. If n has no successor, NEXT(n) returns NULL.

For associative arrays with VARCHAR2 keys, these methods return the appropriate key value; ordering is based on the binary values of the characters in the string, unless the NLS_COMP initialization parameter is set to ANSI, in which case the ordering is based on the locale-specific sort order specified by the NLS_SORT initialization parameter.

These methods are more reliable than looping through a fixed set of subscript values, because elements might be inserted or deleted from the collection during the loop. This is especially true for associative arrays, where the subscripts might not be in consecutive order and so the sequence of subscripts might be (1,2,4,8,16) or ('A','E','I','O','U').

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1966,1971,1984,1989,1999);
BEGIN
   dbms_output.put_line('The element after #2 is #' || n.NEXT(2));
   dbms_output.put_line('The element before #2 is #' || n.PRIOR(2));
   n.DELETE(3); -- Delete an element to show how NEXT can handle gaps.
   dbms_output.put_line('Now the element after #2 is #' || n.NEXT(2));
   IF n.PRIOR(n.FIRST) IS NULL THEN
      dbms_output.put_line('Can''t get PRIOR of the first element or NEXT of the
last.');
   END IF;
END;
/
```

You can use PRIOR or NEXT to traverse collections indexed by any series of subscripts. The following example uses NEXT to traverse a nested table from which some elements have been deleted:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1,3,5,7);
   counter INTEGER;
BEGIN
   n.DELETE(2); -- Delete second element.
-- When the subscripts have gaps, the loop logic is more extensive. We start at
the
-- first element, and keep looking for the next element until there are no more.
   counter := n.FIRST;
```

```
   WHILE counter IS NOT NULL
   LOOP
      dbms_output.put_line('Counting up: Element #' || counter || ' = ' ||
n(counter));
      counter := n.NEXT(counter);
   END LOOP;

-- Run the same loop in reverse order.
   counter := n.LAST;
   WHILE counter IS NOT NULL
   LOOP
      dbms_output.put_line('Counting down: Element #' || counter || ' = ' ||
n(counter));
      counter := n.PRIOR(counter);
   END LOOP;
END;
/
```

When traversing elements, `PRIOR` and `NEXT` skip over deleted elements.

Increasing the Size of a Collection (EXTEND Method)

To increase the size of a nested table or varray, use `EXTEND`.

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.

You cannot use `EXTEND` to add elements to an uninitialized.

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. If `EXTEND` encounters deleted elements, it includes them in its tally. PL/SQL keeps placeholders for deleted elements so that you can re-create them by assigning new values.

```
DECLARE
   TYPE NumList IS TABLE OF INTEGER;
   n NumList := NumList(2,4,6,8);
   x NumList := NumList(1,3);
   PROCEDURE print_numlist(the_list NumList) IS
      output VARCHAR2(128);
   BEGIN
      FOR i IN the_list.FIRST .. the_list.LAST
      LOOP
         output := output || NVL(TO_CHAR(the_list(i)),'NULL') || ' ';
      END LOOP;
      dbms_output.put_line(output);
   END;
BEGIN
   dbms_output.put_line('At first, N has ' || n.COUNT || ' elements.');
   n.EXTEND(5); -- Add 5 elements at the end.
   dbms_output.put_line('Now N has ' || n.COUNT || ' elements.');
-- Elements 5, 6, 7, 8, and 9 are all NULL.
   print_numlist(n);
```

```
   dbms_output.put_line('At first, X has ' || x.COUNT || ' elements.');
   x.EXTEND(4,2); -- Add 4 elements at the end.
   dbms_output.put_line('Now X has ' || x.COUNT || ' elements.');
-- Elements 3, 4, 5, and 6 are copies of element #2.
   print_numlist(x);
END;
/
```

When it includes deleted elements, the internal size of a nested table differs from the values returned by `COUNT` and `LAST`. 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, regardless of position, are treated alike.

Decreasing the Size of a Collection (TRIM Method)

This procedure has two forms:

- `TRIM` removes one element from the end of a collection.
- `TRIM(n)` removes `n` elements from the end of a collection.

For example, this statement removes the last three elements from nested table `courses`:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1,2,3,5,7,11);
   PROCEDURE print_numlist(the_list NumList) IS
      output VARCHAR2(128);
   BEGIN
      IF n.COUNT = 0 THEN
         dbms_output.put_line('No elements in collection.');
      ELSE
         FOR i IN the_list.FIRST .. the_list.LAST
         LOOP
            output := output || NVL(TO_CHAR(the_list(i)),'NULL') || ' ';
         END LOOP;
         dbms_output.put_line(output);
      END IF;
   END;
BEGIN
   print_numlist(n);
   n.TRIM(2); -- Remove last 2 elements.
   print_numlist(n);
   n.TRIM; -- Remove last element.
   print_numlist(n);
   n.TRIM(n.COUNT); -- Remove all remaining elements.
   print_numlist(n);

-- If too many elements are specified, TRIM raises the exception SUBSCRIPT_BEYOND_
COUNT.
   BEGIN
      n := NumList(1,2,3);
      n.TRIM(100);
      EXCEPTION
         WHEN SUBSCRIPT_BEYOND_COUNT THEN
            dbms_output.put_line('I guess there weren''t 100 elements that could
be trimmed.');
   END;
```

```
-- When elements are removed by DELETE, placeholders are left behind. TRIM counts
these
-- placeholders as it removes elements from the end.

   n := NumList(1,2,3,4);
   n.DELETE(3);  -- delete element 3
-- At this point, n contains elements (1,2,4).
-- TRIMming the last 2 elements removes the 4 and the placeholder, not 4 and 2.
   n.TRIM(2);
   print_numlist(n);
END;
/
END;
/
```

If n is too large, `TRIM(n)` raises `SUBSCRIPT_BEYOND_COUNT`.

`TRIM` operates on the internal size of a collection. If `TRIM` encounters deleted elements, it includes them in its tally. 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, COUNT equals 2, the number of valid
      elements remaining. So, you might expect the next
      statement to empty the nested table by trimming
      elements 1 and 2. Instead, it trims valid element 2
      and deleted element 3 because TRIM includes deleted
      elements in its tally. */
   courses.TRIM(courses.COUNT);
   dbms_output.put_line(courses(1));  -- prints 'Biol 4412'
END;
/
```

In general, do not depend on the interaction between `TRIM` and `DELETE`. It is better to treat nested tables like fixed-size arrays and use only `DELETE`, or to treat them like stacks and use only `TRIM` and `EXTEND`.

Because PL/SQL does not keep placeholders for trimmed elements, you cannot replace a trimmed element simply by assigning it a new value.

Deleting Collection Elements (DELETE Method)

This procedure has various forms:

- `DELETE` removes all elements from a collection.
- `DELETE(n)` removes the *n*th element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, `DELETE(n)` does nothing.
- `DELETE(m,n)` removes all elements in the range `m..n` from an associative array or nested table. If m is larger than n or if m or n is null, `DELETE(m,n)` does nothing.

For example:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
```

```
   n NumList := NumList(10,20,30,40,50,60,70,80,90,100);
   TYPE NickList IS TABLE OF VARCHAR2(64) INDEX BY VARCHAR2(32);
   nicknames NickList;
BEGIN
   n.DELETE(2);    -- deletes element 2
   n.DELETE(3,6);  -- deletes elements 3 through 6
   n.DELETE(7,7);  -- deletes element 7
   n.DELETE(6,3);  -- does nothing since 6 > 3

   n.DELETE;      -- deletes all elements

   nicknames('Bob') := 'Robert';
   nicknames('Buffy') := 'Esmerelda';
   nicknames('Chip') := 'Charles';
   nicknames('Dan') := 'Daniel';
   nicknames('Fluffy') := 'Ernestina';
   nicknames('Rob') := 'Robert';

   nicknames.DELETE('Chip'); -- deletes element denoted by this key
   nicknames.DELETE('Buffy','Fluffy'); -- deletes elements with keys in this
alphabetic range
END;
/
```

Varrays always have consecutive subscripts, so you cannot delete individual elements except from the end (by using the `TRIM` method).

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 by assigning it a new value.

`DELETE` lets you maintain sparse nested tables. You can store sparse nested tables in the database, just like any other nested tables.

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

Applying Methods to Collection Parameters

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. You can apply the built-in collection methods (`FIRST`, `LAST`, `COUNT`, and so on) to such parameters. You can create general-purpose subprograms that take collection parameters and iterate through their elements, add or delete elements, and so on.

Note: For varray parameters, the value of `LIMIT` is always derived from the parameter type definition, regardless of the parameter mode.

Avoiding Collection Exceptions

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception. Consider the following example:

```
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 nested 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 NULL; END IF;  -- raises NO_DATA_FOUND           (5)
END;
/
```

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 allowed 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.

The following list shows when a given exception is raised:

Collection Exception	Raised when...
COLLECTION_IS_NULL	you try to operate on an atomically null collection.
NO_DATA_FOUND	a subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	a subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	a subscript is outside the allowed range.
VALUE_ERROR	a subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS_INTEGER range, and the subscript is outside this range.

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. You can replace deleted elements by assigning values to them, without raising NO_DATA_FOUND:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   nums NumList := NumList(10,20,30);  -- initialize table
BEGIN
   nums.DELETE(-1);  -- does not raise SUBSCRIPT_OUTSIDE_LIMIT
   nums.DELETE(3);   -- delete 3rd element
   dbms_output.put_line(nums.COUNT);  -- prints 2
   nums(3) := 30;    -- allowed; does not raise NO_DATA_FOUND
   dbms_output.put_line(nums.COUNT);  -- prints 3
END;
/
```

Packaged collection types and local collection types are never compatible. For example, suppose you want to call the following packaged procedure:

```
CREATE PACKAGE pkg AS
   TYPE NumList IS TABLE OF NUMBER;
   PROCEDURE print_numlist (nums NumList);
END pkg;
/

DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n1  pkg.NumList := pkg.NumList(2,4); -- Type from the package.
   n2 NumList := NumList(6,8);          -- Local type.
```

```
BEGIN
   pkg.print_numlist(n1);
-- The packaged procedure can't accept a value of the local type.
   pkg.print_numlist(n2);  -- Causes a compilation error.
END;
/

DROP PACKAGE pkg;
```

The second procedure call fails, because the packaged and local VARRAY types are incompatible despite their identical definitions.

What Is a PL/SQL Record?

A **record** is a group of related data items stored in **fields**, each with its own name and datatype. You can think of a record as a variable that can hold a table row, or some columns from a table row. The fields correspond to table columns.

The %ROWTYPE attribute lets you declare a record that represents a row in a database table, without listing all the columns. Your code keeps working even after columns are added to the table. If you want to represent a subset of columns in a table, or columns from different tables, you can define a view or declare a cursor to select the right columns and do any necessary joins, and then apply %ROWTYPE to the view or cursor.

Defining and Declaring Records

To create records, you define a RECORD type, then declare records of that type. You can also create or find a table, view, or PL/SQL cursor with the values you want, and use the %ROWTYPE attribute to create a matching record.

You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package. When you define your own RECORD type, you can specify a NOT NULL constraint on fields, or give them default values.

```
DECLARE
-- Declare a record type with 3 fields.
  TYPE rec1_t IS RECORD (field1 VARCHAR2(16), field2 NUMBER, field3 DATE);
-- For any fields declared NOT NULL, we must supply a default value.
  TYPE rec2_t IS RECORD (id INTEGER NOT NULL := -1, name VARCHAR2(64) NOT NULL :=
'[anonymous]');

-- Declare record variables of the types declared above.
  rec1 rec1_t;
  rec2 rec2_t;

-- Declare a record variable that can hold a row from the EMPLOYEES table.
-- The fields of the record automatically match the names and types of the
columns.
-- Don't need a TYPE declaration in this case.
  rec3 employees%ROWTYPE;

-- Or we can mix fields that are table columns with user-defined fields.
  TYPE rec4_t IS RECORD (first_name employees.first_name%TYPE, last_name
employees.last_name%TYPE, rating NUMBER);
  rec4 rec4_t;

BEGIN
-- Read and write fields using dot notation
  rec1.field1 := 'Yesterday';
```

```
  rec1.field2 := 65;
  rec1.field3 := TRUNC(SYSDATE-1);

-- We didn't fill in the NAME field, so it takes the default value declared above.
  dbms_output.put_line(rec2.name);
END;
/
```

To store a record in the database, you can specify it in an INSERT or UPDATE statement, if its fields match the columns in the table:

```
...
```

You can use %TYPE to specify a field type corresponding to a table column type. Your code keeps working even if the column type is changed (for example, to increase the length of a VARCHAR2 or the precision of a NUMBER). The following example defines RECORD types to hold information about a department:

```
DECLARE
-- Best: use %ROWTYPE instead of specifying each column.
-- Using <cursor>%ROWTYPE instead of <table>%ROWTYPE since we only want some
columns.
-- Declaring the cursor doesn't run the query, so no performance hit.
   CURSOR c1 IS SELECT department_id, department_name, location_id FROM
departments;
   rec1 c1%ROWTYPE;

-- Use <column>%TYPE in field declarations to avoid problems if the column types
change.
   TYPE DeptRec2 IS RECORD (dept_id   departments.department_id%TYPE, dept_name
departments.department_name%TYPE, dept_loc departments.location_id%TYPE);
   rec2 DeptRec2;

-- Final technique, writing out each field name and specifying the type directly,
is
-- clumsy and unmaintainable for working with table data. Use only for all-PL/SQL
code.
   TYPE DeptRec3 IS RECORD (dept_id NUMBER, dept_name VARCHAR2(14), dept_loc
VARCHAR2(13));
   rec3 DeptRec3;
BEGIN
   NULL;
END;
/
```

PL/SQL lets you define records that contain objects, collections, and other records (called nested records). However, records cannot be attributes of object types.

Using Records as Procedure Parameters and Function Return Values

Records are easy to process using stored procedures because you can pass just one parameter, instead of a separate parameter for each field. For example, you might fetch a table row from the EMPLOYEES table into a record, then pass that row as a parameter to a function that computed that employee's vacation allowance or some other abstract value. The function could access all the information about that employee by referring to the fields in the record.

The next example shows how to return a record from a function. To make the record type visible across multiple stored functions and stored procedures, declare the record type in a package specification.

```
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
   NULL;
END;
/
```

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions:

```
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;
/
```

Assigning Values to Records

To set all the fields in a record to default values, assign to it an uninitialized record of the same type:

```
DECLARE
   TYPE RecordTyp IS RECORD (field1 NUMBER, field2 VARCHAR2(32) DEFAULT
'something');
   rec1 RecordTyp;
   rec2 RecordTyp;
BEGIN
-- At first, rec1 has the values we assign.
   rec1.field1 := 100; rec1.field2 := 'something else';
-- Assigning an empty record to rec1 resets fields to their default values.
-- Field1 is NULL and field2 is 'something' (because of the DEFAULT clause above).
   rec1 := rec2;
   dbms_output.put_line('Field1 = ' || NVL(TO_CHAR(rec1.field1),'<NULL>') || ',
field2 = ' || rec1.field2);
END;
/
```

You can assign a value to a field in a record using an assignment statement with dot notation:

```
emp_info.last_name := 'Fields';
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once.

You can assign one user-defined record to another if they have the same datatype. Having fields that match exactly is not enough. Consider the following example:

```
DECLARE
-- Two identical type declarations.
   TYPE DeptRec1 IS RECORD ( dept_num  NUMBER(2), dept_name VARCHAR2(14));
   TYPE DeptRec2 IS RECORD ( dept_num  NUMBER(2), dept_name VARCHAR2(14));
   dept1_info DeptRec1;
   dept2_info DeptRec2;
   dept3_info DeptRec2;
BEGIN
-- Not allowed; different datatypes, even though fields are the same.
--      dept1_info := dept2_info;
-- This assignment is OK because the records have the same type.
   dept2_info := dept3_info;
END;
/
```

You can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have the same datatypes:

```
DECLARE
   TYPE RecordTyp IS RECORD (last employees.last_name%TYPE, id employees.employee_
id%TYPE);
   CURSOR c1 IS SELECT last_name, employee_id FROM employees;

-- Rec1 and rec2 have different types. But because rec2 is based on a %ROWTYPE, we
can
-- assign is to rec1 as long as they have the right number of fields and the
fields
-- have the right datatypes.
   rec1 RecordTyp;
   rec2 c1%ROWTYPE;
BEGIN
   SELECT last_name, employee_id INTO rec2 FROM employees WHERE ROWNUM < 2;
   rec1 := rec2;
   dbms_output.put_line('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

You can also use the SELECT or FETCH statement to fetch column values into a record. The columns in the select-list must appear in the same order as the fields in your record.

```
DECLARE
   TYPE RecordTyp IS RECORD (last employees.last_name%TYPE, id employees.employee_
id%TYPE);
   rec1 RecordTyp;
BEGIN
   SELECT last_name, employee_id INTO rec1 FROM employees WHERE ROWNUM < 2;
   dbms_output.put_line('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

You cannot assign a list of values to a record using an assignment statement. There is no constructor-like notation for records.

Comparing Records

Records cannot be tested for nullity, or compared for equality, or inequality.

If you want to make such comparisons, write your own function that accepts two records as parameters and does the appropriate checks or comparisons on the corresponding fields.

Inserting PL/SQL Records into the Database

A PL/SQL-only extension of the INSERT statement lets you insert records into database rows, using a single variable of type RECORD or %ROWTYPE in the VALUES clause instead of a list of fields. That makes your code more readable and maintainable.

If you issue the INSERT through the FORALL statement, you can insert values from an entire collection of records.

The number of fields in the record must equal the number of columns listed in the INTO clause, and corresponding fields and columns must have compatible datatypes. To make sure the record is compatible with the table, you might find it most convenient to declare the variable as the type *table_name*%ROWTYPE.

Example 5–34 Inserting a PL/SQL Record Using %ROWTYPE

This example declares a record variable using a %ROWTYPE qualifier. You can insert this variable without specifying a column list. The %ROWTYPE declaration ensures that the record attributes have exactly the same names and types as the table columns.

```
DECLARE
   dept_info dept%ROWTYPE;
BEGIN
-- deptno, dname, and loc are the table columns.
-- The record picks up these names from the %ROWTYPE.
  dept_info.deptno := 70;
  dept_info.dname := 'PERSONNEL';
  dept_info.loc := 'DALLAS';
-- Using the %ROWTYPE means we can leave out the column list
-- (deptno, dname, loc) from the INSERT statement.
   INSERT INTO dept VALUES dept_info;
END;
/
```

Updating the Database with PL/SQL Record Values

A PL/SQL-only extension of the UPDATE statement lets you update database rows using a single variable of type RECORD or %ROWTYPE on the right side of the SET clause, instead of a list of fields.

If you issue the UPDATE through the FORALL statement, you can update a set of rows using values from an entire collection of records.

Also with an UPDATE statement, you can specify a record in the RETURNING clause to retrieve new values into a record. If you issue the UPDATE through the FORALL statement, you can retrieve new values from a set of updated rows into a collection of records.

The number of fields in the record must equal the number of columns listed in the SET clause, and corresponding fields and columns must have compatible datatypes.

Example 5–35 Updating a Row Using a Record

You can use the keyword ROW to represent an entire row:

```
DECLARE
   dept_info dept%ROWTYPE;
BEGIN
   dept_info.deptno := 30;
   dept_info.dname := 'MARKETING';
   dept_info.loc := 'ATLANTA';
-- The row will have values for the filled-in columns, and null
-- for any other columns.
   UPDATE dept SET ROW = dept_info WHERE deptno = 30;
END;
/
```

The keyword ROW is allowed only on the left side of a SET clause.

The argument to SET ROW must be a real PL/SQL record, not a subquery that returns a single row.

The record can also contain collections or objects.

Example 5–36 Using the RETURNING Clause with a Record

The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into a PL/SQL record variable. This eliminates the need to SELECT the row after an insert or update, or before a delete.

By default, you can use this clause only when operating on exactly one row. When you use bulk SQL, you can use the form RETURNING BULK COLLECT INTO to store the results in one or more collections.

The following example updates the salary of an employee and retrieves the employee's name, job title, and new salary into a record variable:

```
DECLARE
   TYPE EmpRec IS RECORD (last_name employees.last_name%TYPE, salary
employees.salary%TYPE);
   emp_info EmpRec;
   emp_id NUMBER := 100;
BEGIN
   UPDATE employees SET salary = salary * 1.1 WHERE employee_id = emp_id
      RETURNING last_name, salary INTO emp_info;
   dbms_output.put_line('Just gave a raise to ' || emp_info.last_name ||
      ', who now makes ' || emp_info.salary);
   ROLLBACK;
END;
/
```

Restrictions on Record Inserts/Updates

Currently, the following restrictions apply to record inserts/updates:

- Record variables are allowed only in the following places:
  - On the right side of the SET clause in an UPDATE statement
  - In the VALUES clause of an INSERT statement
  - In the INTO subclause of a RETURNING clause

  Record variables are not allowed in a SELECT list, WHERE clause, GROUP BY clause, or ORDER BY clause.

- The keyword ROW is allowed only on the left side of a SET clause. Also, you cannot use ROW with a subquery.
- In an UPDATE statement, only one SET clause is allowed if ROW is used.
- If the VALUES clause of an INSERT statement contains a record variable, no other variable or value is allowed in the clause.
- If the INTO subclause of a RETURNING clause contains a record variable, no other variable or value is allowed in the subclause.
- The following are *not* supported:
  - Nested record types
  - Functions that return a record
  - Record inserts/updates using the EXECUTE IMMEDIATE statement.

Querying Data into Collections of Records

You can use the BULK COLLECT clause with a SELECT INTO or FETCH statement to retrieve a set of rows into a collection of records.

```
DECLARE
   TYPE EmployeeSet IS TABLE OF employees%ROWTYPE;
   underpaid EmployeeSet; -- Holds set of rows from EMPLOYEES table.

   CURSOR c1 IS SELECT first_name, last_name FROM employees;
   TYPE NameSet IS TABLE OF c1%ROWTYPE;
   some_names NameSet; -- Holds set of partial rows from EMPLOYEES table.

BEGIN
-- With one query, we bring all the relevant data into the collection of records.
   SELECT * BULK COLLECT INTO underpaid FROM employees
      WHERE salary < 2500 ORDER BY salary DESC;

-- Now we can process the data by examining the collection, or passing it to
-- a separate procedure, instead of writing a loop to FETCH each row.
   dbms_output.put_line(underpaid.COUNT || ' people make less than 2500.');
   FOR i IN underpaid.FIRST .. underpaid.LAST
   LOOP
      dbms_output.put_line(underpaid(i).last_name || ' makes ' ||
underpaid(i).salary);
   END LOOP;

-- We can also bring in just some of the table columns.
-- Here we get the first and last names of 10 arbitrary employees.
   SELECT first_name, last_name BULK COLLECT INTO some_names FROM employees
      WHERE ROWNUM < 11;
   FOR i IN some_names.FIRST .. some_names.LAST
   LOOP
      dbms_output.put_line('Employee = ' || some_names(i).first_name || ' ' ||
some_names(i).last_name);
   END LOOP;
END;
/
```

6

Performing SQL Operations from PL/SQL

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it. —Samuel Johnson

This chapter shows how PL/SQL supports the SQL commands, functions, and operators that let you manipulate Oracle data.

This chapter contains these topics:

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. It is important not to allow one operation to succeed while the other fails.

At the end of a transaction that makes database changes, Oracle makes all the changes permanent or undoes them all. If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction, restoring the database to its former state.

You use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION 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

For example, the following example shows some queries that call SQL functions:

```
DECLARE
  job_count NUMBER;
  emp_count NUMBER;
BEGIN
  SELECT COUNT(DISTINCT job_id) INTO job_count FROM employees;
  SELECT COUNT(*) INTO emp_count FROM employees;
END;
/
```

SQL Pseudocolumns

PL/SQL recognizes the SQL pseudocolumns: CURRVAL, LEVEL, NEXTVAL, ROWID, and ROWNUM. In PL/SQL, pseudocolumns are only allowed in SQL queries, not in INSERT / UPDATE / DELETE statements, or in other PL/SQL statements such as assignments or conditional tests.

CURRVAL and NEXTVAL

A **sequence** is a schema object that generates sequential numbers. When you create a sequence, you can specify its initial value and an increment. CURRVAL returns the current value in a specified sequence.

Before you can reference CURRVAL in a session, you must use NEXTVAL to generate a number. A reference to NEXTVAL stores the current sequence number in CURRVAL. NEXTVAL increments the sequence and returns the next value. To get the current or next value in a sequence, use dot notation:

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. You can use CURRVAL and NEXTVAL only in a SELECT list, the VALUES clause, and the SET clause. The following example shows how to generate a new sequence number and refer to that same number in more than one statement:

```
CREATE TABLE employees_temp AS SELECT employee_id, first_name FROM employees;
CREATE TABLE employees_temp2 AS SELECT employee_id, first_name FROM employees;

DECLARE
  next_value NUMBER;
BEGIN
-- The NEXTVAL value is the same no matter what table you select from.
  SELECT employees_seq.NEXTVAL INTO next_value FROM dual;
-- You usually use NEXTVAL to create unique numbers when inserting data.
  INSERT INTO employees_temp VALUES (employees_seq.NEXTVAL, 'value 1');
-- If you need to store the same value somewhere else, you use CURRVAL.
  INSERT INTO employees_temp2 VALUES (employees_seq.CURRVAL, 'value 1');
-- Because NEXTVAL values might be referenced by different users and
-- applications, and some NEXTVAL values might not be stored in the
-- database, there might be gaps in the sequence.
END;
/

DROP TABLE employees_temp;
DROP TABLE employees_temp2;
```

Each time you reference the NEXTVAL value of a sequence, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

LEVEL

You use LEVEL with the SELECT CONNECT BY statement to organize rows from a database table into a tree structure. You might use sequence numbers to give each row a unique identifier, and refer to those identifiers from other rows to set up parent-child relationships.

LEVEL returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2, grandchildren are level 3, and so on.

In the START WITH clause, you specify a condition that identifies the root of the tree. You specify the direction in which the query traverses the tree (down from the root or up from the branches) with the PRIOR operator.

ROWID

ROWID returns the rowid (binary address) of a row in a database table. You can use variables of type UROWID to store rowids in a readable format.

When you select or fetch a physical rowid into a UROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to a character string. You can compare the UROWID variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement to identify the latest row fetched from a cursor. For an example, see "Fetching Across Commits" on page 6-34.

ROWNUM

ROWNUM returns a number indicating the order in which a row was selected from a table. The first row selected has a ROWNUM of 1, the second row has a ROWNUM of 2, and so on. If a SELECT statement includes an ORDER BY clause, ROWNUMs are assigned to the retrieved rows before the sort is done; use a subselect (shown in the following example) to get the first *n* sorted rows.

You can use ROWNUM in an UPDATE statement to assign unique values to each row in a table, or in the WHERE clause of a SELECT statement to limit the number of rows retrieved:

```
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
   CURSOR c1 IS SELECT employee_id, salary FROM employees_temp
      WHERE salary > 2000 AND ROWNUM <= 10;  -- 10 arbitrary rows
   CURSOR c2 IS SELECT * FROM
     (SELECT employee_id, salary FROM employees_temp
       WHERE salary > 2000 ORDER BY salary DESC)
     WHERE ROWNUM < 5;  -- first 5 rows, in sorted order
BEGIN
-- Each row gets assigned a different number
  UPDATE employees_temp SET employee_id = ROWNUM;
END;
/

DROP TABLE employees_temp;
```

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 *Oracle Database 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 yield TRUE, FALSE, or NULL. You can use the comparison operators listed below to form predicates. You can combine predicates using the logical operators AND, OR, and NOT.

Operator	Description
ALL	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.
ANY, SOME	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.
BETWEEN	Tests whether a value lies in a specified range.
EXISTS	Returns TRUE if a subquery returns at least one row.
IN	Tests for set membership.
IS NULL	Tests for nulls.
LIKE	Tests whether a character string matches a specified pattern, which can include wildcards.

Set Operators

Set operators combine the results of two queries into one result. INTERSECT returns all distinct rows selected by both queries. MINUS returns all distinct rows selected by the first query but not by the second. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

Row Operators

Row operators return or reference particular rows. ALL retains duplicate rows in the result of a query or in an aggregate expression. DISTINCT eliminates duplicate rows from the result of a query or from an aggregate expression. PRIOR refers to the parent row of the current row returned by a tree-structured query.

Performing DML Operations from PL/SQL (INSERT, UPDATE, and DELETE)

You can write INSERT, UPDATE, and DELETE statements directly in PL/SQL programs, without any special notation:

```
CREATE table1 AS SELECT object_name, object_type FROM user_objects;

BEGIN
   INSERT INTO table1(col1, col2) VALUES('value1','value2');
   UPDATE table1 SET col1 = 'another value' WHERE col2 IS NULL;
   DELETE FROM table1 WHERE col1 = col2;
   COMMIT;
END;
/

DROP table1;
```

To find out how many rows are affected by these statements, you can check the value of SQL%ROWCOUNT:

```
SET SERVEROUTPUT ON;
BEGIN
  UPDATE employees SET salary = salary * 1.05 WHERE ...;
  dbms_output.put_line('Updated ' || SQL%ROWCOUNT || ' salaries.');
END;
/
```

Wherever you would use literal values, or bind variables in some other programming language, you can directly substitute PL/SQL variables:

```
CREATE table1 AS SELECT object_name, object_type FROM user_objects;

DECLARE
   x VARCHAR2(128) := 'value1';
   y NUMBER := 10;
BEGIN
   INSERT INTO table1(col1, col2) VALUES(x, x);
   UPDATE table1 SET col1 = x WHERE col3 < y;
   DELETE FROM table1 WHERE col1 = x;
   COMMIT;
END;
/

DROP table1;
```

With this notation, you can use variables in place of values in the WHERE clause. To use variables in place of table names, column names, and so on, requires the EXECUTE IMMEDIATE statement that is explained in ...

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 with the %BULK_ROWCOUNT Attribute" on page 11-12.

%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). To save an attribute value for later use, assign it to a Boolean variable immediately. Doing other operations, such as procedure calls, might change the value of %NOTFOUND before you can test it.

The %NOTFOUND attribute is not useful in combination with the SELECT INTO statement:

If a SELECT INTO statement fails to return a row, PL/SQL raises the predefined exception NO_DATA_FOUND immediately, interrupting the flow of control before you can check %NOTFOUND.

A SELECT INTO statement that calls a SQL aggregate function always returns a value or a null. After such a statement, the %NOTFOUND attribute is always FALSE, so checking it is unnecessary.

Using PL/SQL Records in SQL INSERT and UPDATE Statements

Instead of listing each field of a PL/SQL record in INSERT and UPDATE statements, you can use PL/SQL records directly. The most convenient technique is to declare the record using a %ROWTYPE attribute, so that it has exactly the same fields as the SQL table:

```
DECLARE
    emp_rec emp%ROWTYPE;
BEGIN
   emp_rec.eno := 1500;
   emp_rec.ename := 'Steven Hill';
   emp_rec.sal := '40000';
-- A %ROWTYPE value can fill in all the row fields.
    INSERT INTO emp VALUES emp_rec;

-- The fields of a %ROWTYPE can completely replace the table columns.
    UPDATE emp SET ROW = emp_rec WHERE eno = 100;
END;
/
```

Although this technique integrates PL/SQL variables and types with SQL DML statements, you cannot use PL/SQL records as bind variables in dynamic SQL statements.

> **See Also:** "What Is a PL/SQL Record?" on page 5-32 for more information about PL/SQL records.

Issuing Queries from PL/SQL

PL/SQL lets you perform queries (SELECT statements in SQL) and access individual fields or entire rows from the result set. Depending on the complexity of the processing that you want to do on the query results, you can use various notations.

Selecting At Most One Row: SELECT INTO Statement

If you expect a query to only return one row, you can write a regular SQL SELECT statement with an additional INTO clause specifying the PL/SQL variable to hold the result:

If the query might return more than one row, but you do not care about values after the first, you can restrict any result set to a single row by comparing the ROWNUM value:

If the query might return no rows at all, use an exception handler to specify any actions to take when no data is found:

If you just want to check whether a condition exists in your data, you might be able to code the query with the COUNT(*) operator, which always returns a number and never raises the NO_DATA_FOUND exception:

Selecting Multiple Rows: BULK COLLECT Clause

If you need to bring a large quantity of data into local PL/SQL variables, rather than looping through a result set one row at a time, you can use the BULK COLLECT clause. When you query only certain columns, you can store all the results for each column in a separate collection variable:

```
SELECT employee_id, last_name, salary FROM employees
  BULK COLLECT INTO all_employee_ids, all_last_names, all_salaries;
```

When you query all the columns of a table, you can store the entire result set in a collection of records, which makes it convenient to loop through the results and refer to different columns:

```
SELECT * FROM employees BULK COLLECT INTO all_employees;
FOR i IN all_employees.FIRST .. all_employees.LAST
LOOP
  ...
END LOOP;
```

This technique can be very fast, but also very memory-intensive. If you use it often, you might be able to improve your code by doing more of the work in SQL:

- If you only need to loop once through the result set, use a FOR loop as described in the following sections. This technique avoids the memory overhead of storing a copy of the result set.
- If you are looping through the result set to scan for certain values or filter the results into a smaller set, do this scanning or filtering in the original query instead. You can add more WHERE clauses in simple cases, or use set operators such as INTERSECT and MINUS if you are comparing two or more sets of results.
- If you are looping through the result set and running another query or a DML statement for each result row, you can probably find a more efficient technique. For queries, look at including subqueries or EXISTS or NOT EXISTS clauses in the original query. For DML statements, look at the FORALL statement, which is much faster than coding these statements inside a regular loop.

Looping Through Multiple Rows: Cursor FOR Loop

Perhaps the most common case of a query is one where you issue the SELECT statement, then immediately loop once through the rows of the result set. PL/SQL lets you use a simple FOR loop for this kind of query:

The iterator variable for the FOR loop does not need to be declared in advance. It is a %ROWTYPE record whose field names match the column names from the query, and that exists only during the loop. When you use expressions rather than explicit column names, use column aliases so that you can refer to the corresponding values inside the loop:

Performing Complicated Query Processing: Explicit Cursors

For full control over query processing, you can use explicit cursors in combination with the OPEN, FETCH, and CLOSE statements.

You might want to specify a query in one place but retrieve the rows somewhere else, even in another subprogram. Or you might want to choose very different query parameters, such as ORDER BY or GROUP BY clauses, depending on the situation. Or you might want to process some rows differently than others, and so need more than a simple loop.

Because explicit cursors are so flexible, you can choose from different notations depending on your needs. The following sections describe all the query-processing features that explicit cursors provide.

Querying Data with PL/SQL

In traditional database programming, you process query results using an internal data structure called a **cursor**. In most situations, PL/SQL can manage the cursor for you, so that code to process query results is straightforward and compact. This section discusses how to process both simple queries where PL/SQL manages everything, and complex queries where you interact with the cursor.

Querying Data with PL/SQL: Implicit Cursor FOR Loop

With PL/SQL, it is very simple to issue a query, retrieve each row of the result into a `%ROWTYPE` record, and process each row in a loop:

- You include the text of the query directly in the `FOR` loop.
- PL/SQL creates a record variable with fields corresponding to the columns of the result set.
- You refer to the fields of this record variable inside the loop. You can perform tests and calculations, display output, or store the results somewhere else.

Here is an example that you can run in SQL*Plus. It does a query to get the name and status of every index that you can access.

```
BEGIN
  FOR item IN
  (
    SELECT object_name, status FROM user_objects WHERE object_type = 'INDEX'
       AND object_name NOT LIKE '%$%'
  )
  LOOP
    dbms_output.put_line('Index = ' || item.object_name ||
      ', Status = ' || item.status);
  END LOOP;
END;
/
```

Before each iteration of the `FOR` loop, PL/SQL fetches into the implicitly declared record.

The sequence of statements inside the loop is executed once for each row that satisfies the query. When you leave the loop, the cursor is closed automatically. The cursor is closed even if you use an `EXIT` or `GOTO` statement to leave the loop before all rows are fetched, or an exception is raised inside the loop.

See also: LOOP Statements on page 13-79

Querying Data with PL/SQL: Explicit Cursor FOR Loops

IIf you need to reference the same query from different parts of the same procedure, you can declare a cursor that specifies the query, and process the results using a FOR loop.

The following PL/SQ block runs two variations of the same query, first finding all the tables you can access, then all the indexes you can access:

```
DECLARE
```

```
  CURSOR c1 IS
    SELECT object_name, status FROM user_objects WHERE object_type = 'TABLE'
       AND object_name NOT LIKE '%$%';
BEGIN
  FOR item IN c1 LOOP
    dbms_output.put_line('Table = ' || item.object_name ||
      ', Status = ' || item.status);
  END LOOP;
END;
/
```

See also: LOOP Statements on page 13-79

Defining Aliases for Expression Values in a Cursor FOR Loop

In a cursor FOR loop, PL/SQL creates a %ROWTYPE record with fields corresponding to columns in the result set. The fields have the same names as corresponding columns in the SELECT list.

The select list might contain an expression, such as a column plus a constant, or two columns concatenated together. If so, use a column alias to give unique names to the appropriate columns.

In the following example, full_name and dream_salary are aliases for expressions in the query:

```
SET SERVEROUTPUT ON;

BEGIN
  FOR item IN
  (
    SELECT
      first_name || ' ' || last_name AS full_name,
      salary * 10 AS dream_salary
    FROM employees
    WHERE ROWNUM <= 5
  )
  LOOP
    dbms_output.put_line(item.full_name || ' dreams of making ' ||
      item.dream_salary);
  END LOOP;
END;
/
```

Overview of Explicit Cursors

When you need precise control over query processing, you can explicitly declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.

You use three 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.

This technique requires more code than other techniques such as the implicit cursor FOR loop. Its advantage is flexibility. You can:

- Process several queries in parallel by declaring and opening multiple cursors.

- Process multiple rows in a single loop iteration, skip rows, or split the processing into more than one loop.

Declaring a Cursor

You must declare a cursor before referencing it in other statements. You give the cursor a name and associate it with a specific query. You can optionally declare a return type for the cursor (such as *table_name*%ROWTYPE). You can optionally specify parameters that you use in the WHERE clause instead of referring to local variables. These parameters can have default values.

For example, you might declare cursors like these:

```
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 is not a PL/SQL variable: you cannot assign values to a cursor or use it in an expression. Cursors and variables follow the same scoping rules. Naming cursors after database tables is possible but not recommended.

A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be IN parameters; they supply values in the query, but do not return any values from the query. You cannot impose the constraint NOT NULL on a cursor parameter.

As the example below shows, you can initialize cursor parameters to default values. You can pass different numbers of actual parameters to a cursor, accepting or overriding the default values as you please. Also, you can add new formal parameters without having to change existing references to the cursor.

```
DECLARE
   CURSOR c1 (low  INTEGER DEFAULT 0,
              high INTEGER DEFAULT 99) IS SELECT ...
```

Cursor parameters can be referenced only within the query specified in the cursor declaration. The parameter values are used by the associated query when the cursor is opened.

Opening a Cursor

Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria. For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. An example of the OPEN statement follows:

```
DECLARE
   CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;
   ...
BEGIN
   OPEN c1;
   ...
END;
```

Rows in the result set are retrieved by the FETCH statement, not when the OPEN statement is executed.

Fetching with a Cursor

Unless you use the BULK COLLECT clause (discussed in the next section), the FETCH statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and advances the cursor to the next row in the result set.

You can store each column in a separate variable, or store the entire row in a record that has the appropriate fields (usually declared using %ROWTYPE):

```
-- This cursor queries 3 columns.
-- Each column is fetched into a separate variable.
FETCH c1 INTO my_empno, my_ename, my_deptno;
-- This cursor was declared as SELECT * FROM employees.
-- An entire row is fetched into the my_employees record, which
-- is declared with the type employees%ROWTYPE.
FETCH c2 INTO my_employees;
```

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:

```
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. 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:

```
DECLARE
  my_sal employees.salary%TYPE;
  my_job employees.job_id%TYPE;
  factor INTEGER := 2;
  CURSOR c1 IS
    SELECT factor*salary FROM employees WHERE job_id = my_job;
BEGIN
   OPEN c1;  -- 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:

```
DECLARE
   CURSOR c1 IS SELECT last_name FROM employees ORDER BY last_name;
   name1 employees.last_name%TYPE;
   name2 employees.last_name%TYPE;
   name3 employees.last_name%TYPE;
BEGIN
   OPEN c1;
```

```
   FETCH c1 INTO name1;  -- this fetches first row
   FETCH c1 INTO name2;  -- this fetches second row
   FETCH c1 INTO name3;  -- this fetches third row
   CLOSE c1;
END;
/
```

If you fetch past the last row in the result set, the values of the target variables are undefined.

Note: Eventually, the FETCH statement fails to return a row. When that happens, no exception is raised. To detect the failure, use the cursor attribute %FOUND or %NOTFOUND. For more information, see "Using Cursor Expressions" on page 6-27.

Fetching Bulk Data with a Cursor

The BULK COLLECT clause lets you fetch all rows from the result set at once (see "Retrieving Query Results into Collections with the BULK COLLECT Clause" on page 11-15). In the following example, you bulk-fetch from a cursor into two collections:

```
DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;
  nums  NumTab;
  names NameTab;
  CURSOR c1 IS
    SELECT employee_id, last_name
      FROM employees
      WHERE job_id = 'ST_CLERK';
BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO nums, names;
-- Here is where you iterate through the elements in the NUMS and
-- NAMES collections.
  NULL;
  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, which runs the query again with the latest values of any cursor parameters and variables referenced in the WHERE clause. Any other operation on a closed cursor raises the predefined exception INVALID_CURSOR.

Using Subqueries

A **subquery** is a query (usually enclosed by parentheses) that appears within another SQL data manipulation statement. The statement acts upon the single value or set of values returned by the subquery. For example:

- You can use a subquery to find the MAX(), MIN(), or AVG() value for a column, and use that single value in a comparison in a WHERE clause.
- You can use a subquery to find a set of values, and use this values in an IN or NOT IN comparison in a WHERE clause. This technique can avoid joins.

- You can filter a set of values with a subquery, and apply other operations like ORDER BY and GROUP BY in the outer query.
- You can use a subquery in place of a table name, in the FROM clause of a query. This technique lets you join a table with a small set of rows from another table, instead of joining the entire tables.
- You can create a table or insert into a table, using a set of rows defined by a subquery.

```
DECLARE
  CURSOR c1 IS
-- The main query returns only rows where the salary is greater than the average
salary.
    SELECT employee_id, last_name FROM employees WHERE salary > (SELECT
AVG(salary) FROM employees);

  CURSOR c2 IS
-- The subquery returns all the rows in descending order of salary.
-- The main query returns just the top 10 highest-paid employees.
    SELECT * FROM
      (SELECT last_name, salary FROM employees ORDER BY salary DESC, last_name)
    WHERE ROWNUM < 11;
BEGIN
  FOR person IN c1
  LOOP
    dbms_output.put_line('Above-average salary: ' || person.last_name);
  END LOOP;
  FOR person IN c2
  LOOP
    dbms_output.put_line('Highest paid: ' || person.last_name || ' $' ||
person.salary);
  END LOOP;
-- The subquery identifies a set of rows to use with CREATE TABLE or INSERT.
  EXECUTE IMMEDIATE
    'CREATE TABLE temp AS (SELECT * FROM employees WHERE salary > 5000)';
  EXECUTE IMMEDIATE 'DROP TABLE temp';
END;
/
```

Using a subquery in the FROM clause, the following query returns the number and name of each department with five or more employees:

```
DECLARE
  CURSOR c1 IS
    SELECT t1.department_id, department_name, staff
      FROM departments t1,
      (
        SELECT department_id, COUNT(*) as staff
          FROM employees
          GROUP BY department_id
      ) t2
    WHERE
      t1.department_id = t2.department_id
      AND staff >= 5;
BEGIN
   FOR dept IN c1
   LOOP
     dbms_output.put_line('Department = ' || dept.department_name ||
       ', staff = ' || dept.staff);
   END LOOP;
```

```
END;
/
```

Using Correlated Subqueries

While a subquery is evaluated only once for each table, a **correlated subquery** is evaluated once for each row. The following example returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the table, the correlated subquery computes the average salary for the corresponding epartment.

```
DECLARE
-- For each department, we find the average salary.
-- Then we find all the employees in that department making
-- more than that average salary.
  CURSOR c1 IS
    SELECT department_id, last_name, salary
      FROM employees t
    WHERE
      salary >
      (
        SELECT AVG(salary)
          FROM employees
        WHERE
          t.department_id = department_id
      )
    ORDER BY department_id;
BEGIN
  FOR person IN c1
  LOOP
    dbms_output.put_line('Making above-average salary = ' ||
       person.last_name);
  END LOOP;
END;
/
```

Writing Maintainable PL/SQL Queries

Instead of referring to local variables, you can declare a cursor that accepts parameters, and pass values for those parameters when you open the cursor. If the query is usually issued with certain values, you can make those values the defaults. You can use either positional notation or named notation to pass the parameter values.

Example 6–1 Passing Parameters to a Cursor FOR Loop

The following example computes the total wages paid to employees in a specified department.

```
DECLARE
  CURSOR c1 (name VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees WHERE last_name = name and salary < max_wage;
BEGIN
  FOR person IN c1('Austin', 30000)
  LOOP
     -- process data record
    dbms_output.put_line('Name = ' || person.last_name ||
      ', salary = ' || person.salary);
```

```
  END LOOP;
END;
/
```

Example 6–2 Passing Parameters to Explicit Cursors

For example, here are several ways to open a cursor:

```
DECLARE
  emp_name employees.last_name%TYPE := 'Austin';
  emp_salary   employees.salary%TYPE := 30000;
  my_record employees%ROWTYPE;
  CURSOR c1 (name VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees WHERE last_name = name and salary < max_wage;
BEGIN
-- Any of the following statements opens the cursor:
-- OPEN c1('Austin', 3000);
-- OPEN c1('Austin', emp_salary);
-- OPEN c1(emp_name, 3000);
-- OPEN c1(emp_name, emp_salary);

  OPEN c1(emp_name, emp_salary);
  LOOP
     FETCH c1 INTO my_record;
     EXIT WHEN c1%NOTFOUND;
     -- process data record
    dbms_output.put_line('Name = ' || my_record.last_name ||
      ', salary = ' || my_record.salary);
  END LOOP;
END;
/
```

To avoid confusion, use different names for cursor parameters and the PL/SQL variables that you pass into those parameters.

Formal parameters declared with a default value do not need a corresponding actual parameter. If you omit them, they assume their default values when the `OPEN` statement is executed.

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 returns NULL. After any fetches, it returns TRUE if the last fetch returned a row, or FALSE if the last fetch did not return a row. The following example uses %FOUND to select an action:

```
DECLARE
   CURSOR c1 IS SELECT last_name, salary FROM employees WHERE ROWNUM < 11;
   my_ename employees.last_name%TYPE;
   my_salary employees.salary%TYPE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_ename, my_salary;
      IF c1%FOUND THEN  -- fetch succeeded
         dbms_output.put_line('Name = ' || my_ename || ', salary = ' ||
            my_salary);
      ELSE  -- fetch failed, so exit loop
         EXIT;
      END IF;
   END LOOP;
END;
/
```

If a cursor or cursor variable is not open, referencing it with %FOUND raises the predefined exception INVALID_CURSOR.

%ISOPEN Attribute: Is the Cursor Open?

%ISOPEN returns TRUE if its cursor or cursor variable is open; otherwise, %ISOPEN returns FALSE. The following example uses %ISOPEN to select an action:

```
DECLARE
   CURSOR c1 IS SELECT last_name, salary FROM employees WHERE ROWNUM < 11;
   the_name employees.last_name%TYPE;
   the_salary employees.salary%TYPE;
BEGIN
   IF c1%ISOPEN = FALSE THEN  -- cursor was not already open
      OPEN c1;
   END IF;
   FETCH c1 INTO the_name, the_salary;
   CLOSE c1;
END;
/
```

%NOTFOUND Attribute: Has a Fetch Failed?

%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields FALSE if the last fetch returned a row, or TRUE if the last fetch failed to return a row. In the following example, you use %NOTFOUND to exit a loop when FETCH fails to return a row:

```
DECLARE
   CURSOR c1 IS SELECT last_name, salary FROM employees WHERE ROWNUM < 11;
   my_ename employees.last_name%TYPE;
   my_salary employees.salary%TYPE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_ename, my_salary;
      IF c1%NOTFOUND THEN -- fetch failed, so exit loop
-- A shorter form of this test is "EXIT WHEN c1%NOTFOUND;"
```

```
         EXIT;
      ELSE   -- fetch succeeded
         dbms_output.put_line('Name = ' || my_ename || ', salary = ' ||
            my_salary);
      END IF;
   END LOOP;
END;
/
```

Before the first fetch, %NOTFOUND returns NULL. If FETCH never executes successfully, the loop is never exited, because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement instead:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If a cursor or cursor variable is not open, referencing it with %NOTFOUND raises an INVALID_CURSOR exception.

%ROWCOUNT Attribute: How Many Rows Fetched So Far?

When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. The following example uses %ROWCOUNT to test if more than ten rows have been fetched:

```
DECLARE
   CURSOR c1 IS SELECT last_name FROM employees WHERE ROWNUM < 11;
   name employees.last_name%TYPE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO name;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line(c1%ROWCOUNT || '. ' || name);
      IF c1%ROWCOUNT = 5 THEN
         dbms_output.put_line('--- Fetched 5th record ---');
      END IF;
   END LOOP;
   CLOSE c1;
END;
/
```

If a cursor or cursor variable is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR.

Table 6–1 shows what each cursor attribute returns before and after you execute an OPEN, FETCH, or CLOSE statement.

Table 6–1 Cursor Attribute Values

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	before	exception	FALSE	exception	exception
	after	NULL	TRUE	NULL	0
First FETCH	before	NULL	TRUE	NULL	0
	after	TRUE	TRUE	FALSE	1
Next FETCH(es)	before	TRUE	TRUE	FALSE	1

Table 6–1 (Cont.) Cursor Attribute Values

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
	after	TRUE	TRUE	FALSE	data dependent
Last FETCH	before	TRUE	TRUE	FALSE	data dependent
	after	FALSE	TRUE	TRUE	data dependent
CLOSE	before	FALSE	TRUE	TRUE	data dependent
	after	exception	FALSE	exception	exception

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.

Using Cursor Variables (REF CURSORs)

Like a cursor, a cursor variable points to the current row in the result set of a multi-row query. A cursor variable is more flexible because it is not tied to a specific query. You can open a cursor variable for any query that returns the right set of columns.

You pass a cursor variable as a parameter to local and stored subprograms. Opening the cursor variable in one subprogram, and processing it in a different subprogram, helps to centralize data retrieval. This technique is also useful for multi-language applications, where a PL/SQL subprogram might return a result set to a subprogram written in a different language.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as an input host variable (bind variable) to PL/SQL. Application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side. Or, you can pass cursor variables back and forth between a client and the database server through remote procedure calls.

What Are Cursor Variables (REF CURSORs)?

Cursor variables are like pointers to result sets. You use them when you want to perform a query in one subprogram, and process the results in a different subprogram (possibly one written in a different language). A cursor variable has datatype REF CURSOR, and you might see them referred to informally as REF CURSORs.

Unlike an explicit cursor, which always refers to the same query work area, a cursor variable can refer to different work areas. You cannot use a cursor variable where a cursor is expected, or vice versa.

Why Use Cursor Variables?

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. PL/SQL and its clients share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle database server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it, as you pass the value of a cursor variable from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side. You can also reduce network traffic by having a PL/SQL block open or close several host cursor variables in a single round trip.

Declaring REF CURSOR Types and Cursor Variables

To create cursor variables, you define a REF CURSOR type, then declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package. In the following example, you declare a REF CURSOR type that represents a result set from the DEPARTMENTS table:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
```

REF CURSOR types can be *strong* (with a return type) or *weak* (with no return type).

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with queries that return the right set of columns. Weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Because there is no type checking with a weak REF CURSOR, all such types are interchangeable. Instead of creating a new type, you can use the predefined type SYS_REFCURSOR.

Once you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram.

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;  -- strong
   TYPE GenericCurTyp IS REF CURSOR;  -- weak
   cursor1 EmpCurTyp;
   cursor2 GenericCurTyp;
   my_cursor SYS_REFCURSOR; -- didn't need to declare a new type above
```

The following example declares the cursor variable dept_cv:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

To avoid declaring the same REF CURSOR type in each subprogram that uses it, you can put the REF CURSOR declaration in a package spec. You can declare cursor variables of that type in the corresponding package body, or within your own procedure or function.

Example 6–3 Cursor Variable Returning %ROWTYPE

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to refer to a strongly typed cursor variable:

```
DECLARE
   TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   tmp_cv TmpCurTyp;  -- declare cursor variable
   TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
   emp_cv EmpCurTyp;  -- declare cursor variable
BEGIN
  NULL;
END;
```

```
/
```

Example 6–4 Cursor Variable Returning %TYPE

You can also use %TYPE to provide the datatype of a record variable:

```
DECLARE
   dept_rec departments%ROWTYPE;  -- declare record variable
   TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
BEGIN
  NULL;
END;
/
```

Example 6–5 Cursor Variable Returning Record Type

This example specifies a user-defined RECORD type in the RETURN clause:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      employee_id NUMBER,
      last_name VARCHAR2(30),
      salary   NUMBER(7,2));
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   emp_cv EmpCurTyp;  -- declare cursor variable
BEGIN
  NULL;
END;
/
```

Passing Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of functions and procedures. The following example defines a REF CURSOR type, then declares a cursor variable of that type as a formal parameter:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   emp EmpCurTyp;

-- Once we have a result set, we can process all the rows
-- inside a single procedure rather than calling a procedure
-- for each row.
   PROCEDURE process_emp_cv (emp_cv IN EmpCurTyp) IS
      person employees%ROWTYPE;
   BEGIN
      dbms_output.put_line('-----');
      dbms_output.put_line('Here are the names from the result set:');
      LOOP
         FETCH emp_cv INTO person;
         EXIT WHEN emp_cv%NOTFOUND;
         dbms_output.put_line('Name = ' || person.first_name ||
            ' ' || person.last_name);
      END LOOP;
   END;

BEGIN
-- First find 10 arbitrary employees.
```

```
  OPEN emp FOR SELECT * FROM employees WHERE ROWNUM < 11;
  process_emp_cv(emp);
  CLOSE emp;
-- Then find employees matching a condition.
  OPEN emp FOR SELECT * FROM employees WHERE last_name LIKE 'R%';
  process_emp_cv(emp);
  CLOSE emp;
END;
/
```

Note: Like all pointers, cursor variables increase the possibility of parameter aliasing. See "Overloading Subprogram Names" on page 8-9.

Controlling Cursor Variables: OPEN-FOR, FETCH, and CLOSE

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.

```
OPEN {cursor_variable | :host_cursor_variable} FOR
{  select_statement
 | dynamic_string [USING bind_argument[, bind_argument]...] };
```

The cursor variable can be declared directly in PL/SQL, or in a PL/SQL host environment such as an OCI program.

The `SELECT` statement for the query can be coded directly in the statement, or can be a string variable or string literal. When you use a string as the query, it can include placeholders for bind variables, and you specify the corresponding values with a USING clause.

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 "OPEN-FOR-USING Statement" on page 13-97.

Unlike cursors, cursor variables take no parameters. Instead, you can pass whole queries (not just parameters) to a cursor variable. The query can reference host variables and PL/SQL variables, parameters, and functions.

The example below opens a cursor variable. Notice that you can apply cursor attributes (`%FOUND`, `%NOTFOUND`, `%ISOPEN`, and `%ROWCOUNT`) to a cursor variable.

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   emp_cv EmpCurTyp;
BEGIN
   IF NOT emp_cv%ISOPEN THEN
      /* Open cursor variable. */
      OPEN emp_cv FOR SELECT * FROM employees;
   END IF;
   CLOSE emp_cv;
END;
/
```

Other `OPEN-FOR` statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. (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.

Example 6–6 Stored Procedure to Open a Ref Cursor

Typically, you open a cursor variable by passing it to a stored procedure that declares an IN OUT parameter that is a cursor variable. For example, the following procedure opens a cursor variable:

```
CREATE PACKAGE emp_data AS
  TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp);
END emp_data;
/

CREATE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/

DROP PACKAGE emp_data;
```

You can also use a standalone stored procedure to open the cursor variable. Define the REF CURSOR type in a package, then reference that type in the parameter declaration for the stored procedure.

Example 6–7 Stored Procedure to Open Ref Cursors with Different Queries

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.

```
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;

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;
```

Example 6–8 Cursor Variable with Different Return Types

For more flexibility, a stored procedure can execute queries with different return types:

```
CREATE PACKAGE admin_data AS
   TYPE GenCurTyp IS REF CURSOR;
```

```
   PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT);
END admin_data;

CREATE PACKAGE BODY admin_data AS
   PROCEDURE open_cv (generic_cv IN OUT GenCurTyp, choice INT) IS
   BEGIN
      IF choice = 1 THEN
         OPEN generic_cv FOR SELECT * FROM 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 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.

Fetching from a Cursor Variable

The FETCH statement retrieves rows from the result set of a multi-row query. It works the same with cursor variables as with explicit cursors.

Example 6–9 Fetching from a Cursor Variable into a Record

The following example fetches rows one at a time from a cursor variable into a record:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   emp_cv EmpCurTyp;
```

```
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_cv FOR SELECT * FROM employees WHERE salary < 3000;
   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
      dbms_output.put_line('Name = ' || emp_rec.first_name || ' ' ||
         emp_rec.last_name);
   END LOOP;
   CLOSE emp_cv;
END;
/
```

Example 6–10 Fetching from a Cursor Variable into Collections

Using the BULK COLLECT clause, you can bulk fetch rows from a cursor variable into one or more collections:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   TYPE NameList IS TABLE OF employees.last_name%TYPE;
   TYPE SalList IS TABLE OF employees.salary%TYPE;
   emp_cv EmpCurTyp;
   names  NameList;
   sals   SalList;
BEGIN
   OPEN emp_cv FOR SELECT last_name, salary FROM employees WHERE salary < 3000;
   FETCH emp_cv BULK COLLECT INTO names, sals;
   CLOSE emp_cv;
-- Now loop through the NAMES and SALS collections.
   FOR i IN names.FIRST .. names.LAST
   LOOP
      dbms_output.put_line('Name = ' || names(i) || ', salary = ' ||
         sals(i));
   END LOOP;
END;
/
```

Any variables in the associated query are evaluated only when the cursor variable is opened. To change the result set or the values of variables in the query, reopen the cursor variable with the variables set to new values. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set.

PL/SQL makes sure the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. If there is a mismatch, an error occurs at compile time if the cursor variable is strongly typed, or at run time if it is weakly typed. At run time, PL/SQL raises the predefined exception ROWTYPE_MISMATCH *before* the first fetch. If you trap the error and execute the FETCH statement using a different (compatible) INTO clause, no rows are lost.

When you declare a cursor variable as the formal parameter of a subprogram that fetches from the cursor variable, you must specify the IN or IN OUT mode. If the subprogram also opens the cursor variable, you must specify the IN OUT mode.

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Closing a Cursor Variable

The CLOSE statement disables a cursor variable and makes the associated result set undefined. Close the cursor variable after the last row is processed.

When declaring a cursor variable as the formal parameter of a subprogram that closes the cursor variable, you must specify the IN or IN OUT mode.

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Reducing Network Traffic When Passing Host Cursor Variables to PL/SQL

When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping OPEN-FOR statements. For example, the following PL/SQL block opens multiple cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
   OPEN :emp_cv FOR SELECT * FROM employees;
   OPEN :dept_cv FOR SELECT * FROM departments;
   OPEN :loc_cv FOR SELECT * FROM locations;
END;
```

This technique might be useful in Oracle Forms, for 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, so your OCI or Pro*C program can use these work areas for ordinary cursor operations. In the following example, you open several such work areas in a single round trip:

```
BEGIN
   OPEN :c1 FOR SELECT 1 FROM dual;
   OPEN :c2 FOR SELECT 1 FROM dual;
   OPEN :c3 FOR SELECT 1 FROM dual;
END;
```

The cursors assigned to c1, c2, and c3 behave normally, and you can use them for any purpose. When finished, release the cursors as follows:

```
BEGIN
   CLOSE :c1;
   CLOSE :c2;
   CLOSE :c3;
END;
```

Avoiding Errors with Cursor Variables

If both cursor variables involved in an assignment are strongly typed, they must have exactly the same datatype (not just the same return type). If one or both cursor variables are weakly typed, they can have different datatypes.

If you try to fetch from, close, or refer to cursor attributes of a cursor variable that does not point to a query work area, PL/SQL raises the INVALID_CURSOR exception. You can make a cursor variable (or parameter) point to a query work area in two ways:

- OPEN the cursor variable FOR the query.
- Assign to the cursor variable the value of an already OPENed host cursor variable or PL/SQL cursor variable.

If you assign an unopened cursor variable to another cursor variable, the second one remains invalid even after you open the first one.

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises ROWTYPE_MISMATCH if the return types of the actual and formal parameters are incompatible.

Restrictions on Cursor Variables

Currently, cursor variables are subject to the following restrictions:

- You cannot declare cursor variables in a package spec. For example, the following declaration is not allowed:

```
CREATE PACKAGE emp_stuff AS
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   emp_cv EmpCurTyp;  -- not allowed
END emp_stuff;
```

- You cannot pass cursor variables to a procedure that is called through a database link.
- 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.
- Database columns cannot store the values of cursor variables. There is no equivalent type to use in a CREATE TABLE statement.
- You cannot store cursor variables in an associative array, nested table, or varray.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected. For example, you cannot reference a cursor variable in a cursor FOR loop.

Using Cursor Expressions

A cursor expression returns a nested cursor. Each row in the result set can contain values as usual, plus cursors produced by subqueries involving the other values in the row. A single query can return a large set of related values retrieved from multiple tables. You can process the result set with nested loops that fetch first from the rows of the result set, then from any nested cursors within those rows.

PL/SQL supports queries with cursor expressions as part of cursor declarations, REF CURSOR declarations and ref cursor variables. You can also use cursor expressions in dynamic SQL queries. Here is the syntax:

```
CURSOR(subquery)
```

A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is 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.

```
DECLARE
   TYPE emp_cur_typ IS REF CURSOR;
   emp_cur emp_cur_typ;
   dept_name departments.department_name%TYPE;
   emp_name employees.last_name%TYPE;
   CURSOR c1 IS SELECT
      department_name,
-- The 2nd item in the result set is another result set,
-- which is represented as a ref cursor and labelled "employees".
      CURSOR
      (
         SELECT e.last_name FROM employees e
         WHERE e.department_id = d.department_id
      ) employees
   FROM departments d
      WHERE department_name like 'A%';

BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO dept_name, emp_cur;
      EXIT WHEN c1%NOTFOUND;
      dbms_output.put_line('Department: ' || dept_name);
-- For each row in the result set, we can process the result
-- set from a subquery. We could pass the ref cursor to a procedure
-- instead of processing it here in the loop.
      LOOP
         FETCH emp_cur INTO emp_name;
         EXIT WHEN emp_cur%NOTFOUND;
         dbms_output.put_line('   Employee: ' || emp_name);
      END LOOP;
   END LOOP;
   CLOSE c1;
END;
/
```

Constructing REF CURSORs with Cursor Subqueries

You can use cursor subqueries, also know as cursor expressions, to pass sets of rows as parameters to functions. For example, this statement passes a parameter to the StockPivot function consisting of a `REF CURSOR` that represents the rows returned by the cursor subquery:

```
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));
```

Cursor subqueries are often used with table functions, which are explained in "Setting Up Transformation Pipelines with Table Functions" on page 11-28.

Overview of Transaction Processing in PL/SQL

This section explains how to do transaction processing with PL/SQL.

You should already be familiar with the idea of transactions, and how to ensure the consistency of a database, such as the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements. These are Oracle features, available through all programming languages, that let multiple users work on the database concurrently, and ensure that each user sees a consistent version of data and that all changes are applied in the right order.

You usually do not need to write extra code to prevent problems with multiple users accessing data concurrently. Oracle uses **locks** to control concurrent access to data, and locks only the minimum amount of data necessary, for as little time as possible. You can request locks on tables or rows if you really do need this level of control. You can choose from several **modes** of locking such as **row share** and **exclusive**.

Using COMMIT, SAVEPOINT, and ROLLBACK in PL/SQL

You can include `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements directly in your PL/SQL programs.

The `COMMIT` statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users.

The `ROLLBACK` statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

`SAVEPOINT` names and marks the current point in the processing of a transaction. Savepoints let you roll back part of a transaction instead of the whole transaction.

Consider a transaction that transfers money from one bank account to another. It is important that the money come out of one account, and into the other, at exactly the same moment. Otherwise, a problem partway through might make the money be lost from both accounts or be duplicated in both accounts.

```
BEGIN
   UPDATE accts SET bal = my_bal - debit
      WHERE acctno = 7715;
   UPDATE accts SET bal = my_bal + credit
      WHERE acctno = 7720;
   COMMIT WORK;
END;
```

Transactions are not tied to PL/SQL BEGIN-END blocks. A block can contain multiple transactions, and a transaction can span multiple blocks.

The optional COMMENT clause lets you specify a comment to be associated with a distributed transaction. 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:

```
COMMIT COMMENT 'In-doubt order transaction; notify Order Entry';
```

PL/SQL does not support the FORCE clause of SQL, which manually commits an in-doubt distributed transaction.

The following example inserts information about an employee into three different database tables. If an INSERT statement tries to store a duplicate employee number, the predefined exception DUP_VAL_ON_INDEX is raised. To make sure that changes to all three tables are undone, the exception handler executes a ROLLBACK.

```
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.

The following example marks 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
   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. The savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent, but you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers. Reusing a savepoint name within a transaction moves the savepoint from its old position to the current point in the transaction. Thus, a rollback to the savepoint affects only the current part of your transaction:

```
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.

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 transaction. 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, and does not do any rollback.

Ending Transactions

You should explicitly commit or roll back every transaction. Whether you issue the commit or rollback in your PL/SQL program or from a client program depends on the application logic. If you do not commit or roll back a transaction explicitly, the client environment determines its final state.

For example, in the SQL*Plus environment, if your PL/SQL block does not include a COMMIT or ROLLBACK statement, the final state of your transaction depends on what you do after running the block. If you execute a data definition, data control, or COMMIT statement or if you issue the EXIT, DISCONNECT, or QUIT command, Oracle commits the transaction. If you execute a ROLLBACK statement or abort the SQL*Plus session, Oracle rolls back the transaction.

Oracle precompiler programs roll back the transaction unless the program explicitly commits or rolls back work, and disconnects using the RELEASE parameter:

```
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 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 a store manager uses 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;
   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. Queries cannot be FOR UPDATE.

Overriding Default Locking

By default, Oracle locks data structures for you automatically, which is a major strength of the Oracle database: different applications can read and write to the same data without harming each other's data or coordinating with each other.

You can request data locks on specific rows or entire tables if you need to override default locking. Explicit locking lets you deny access to data for the duration of a transaction.:

- With the LOCK TABLE statement, you can explicitly lock entire tables.
- With the SELECT FOR UPDATE statement, you can explicitly lock specific rows of a table to make sure they do not change after you have read them. That way, you can check which or how many rows will be affected by an UPDATE or DELETE statement before issuing the statement, and no other application can change the rows in the meantime.

Using FOR UPDATE

When you declare a cursor that will be referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks. An example follows:

```
DECLARE
   CURSOR c1 IS SELECT 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. Since the rows are no longer locked, you cannot fetch from a FOR UPDATE cursor after a commit. (For a workaround, see "Fetching Across Commits" on page 6-34.)

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:

```
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:

```
DECLARE
   CURSOR c1 IS SELECT empno, job, sal FROM emp FOR UPDATE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO ...
      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.. 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 **Oracle Database 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

PL/SQL raises an exception if you try to fetch from a FOR UPDATE cursor after doing a commit. The FOR UPDATE clause locks the rows when you open the cursor, and unlocks them when you commit.

```
DECLARE
   CURSOR c1 IS SELECT ename FROM emp FOR UPDATE OF sal;
BEGIN
   FOR emp_rec IN c1 LOOP  -- FETCH fails on the second iteration
      INSERT INTO temp VALUES ('still going');
      COMMIT;  -- releases locks
   END LOOP;
END;
```

If you want to fetch across commits, use the ROWID pseudocolumn to mimic the CURRENT OF clause. Select the rowid of each row into a UROWID variable, then use the rowid to identify the current row during subsequent updates and deletes:

```
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;
```

Because the fetched rows are *not* locked by a FOR UPDATE clause, other users might unintentionally overwrite your changes. The extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates.

The next example shows that you can use the %ROWTYPE attribute with cursors that reference the ROWID pseudocolumn:

```
DECLARE
   CURSOR c1 IS SELECT 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;
```

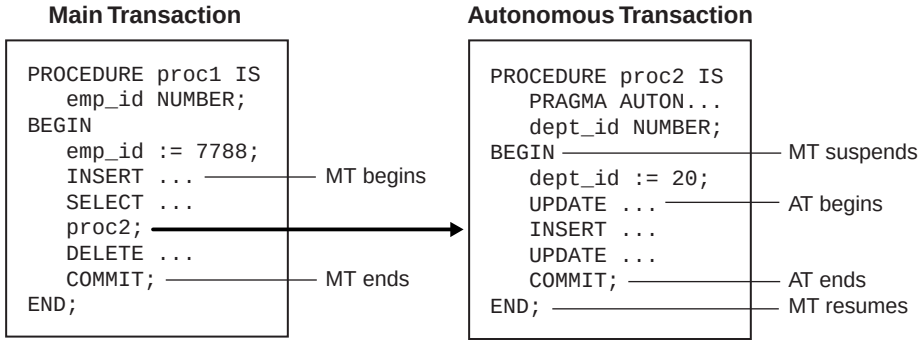
```
      END IF;
   END LOOP;
   CLOSE c1;
END;
```

Doing Independent Units of Work with Autonomous Transactions

An **autonomous transaction** is an independent transaction started by another transaction, the **main transaction**. Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction. For example, if you write auditing data to a log table, you want to commit the audit data even if the operation you are auditing later fails; if something goes wrong recording the audit data, you do not want the main operation to be rolled back.

Figure 6–1 shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

Figure 6–1 Transaction Control Flow



Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

More important, autonomous transactions help you build modular, reusable software components. You can encapsulate autonomous transactions within stored procedures. A calling application does not need to know whether operations done by that stored procedure succeeded or failed.

Defining Autonomous Transactions

To define autonomous transactions, you use the pragma (compiler directive) `AUTONOMOUS_TRANSACTION`. The pragma instructs the PL/SQL compiler to mark a routine as *autonomous* (independent). In this context, the term *routine* includes

- Top-level (not nested) anonymous PL/SQL blocks
- Local, standalone, and packaged 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.

The next example marks a standalone procedure as autonomous:

```
CREATE PROCEDURE close_account (acct_id INTEGER, OUT balance) AS
   PRAGMA AUTONOMOUS_TRANSACTION;
   my_bal REAL;
BEGIN ... END;
```

The following example marks 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.

The example below marks 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;
```

Comparison of Autonomous Transactions and Nested Transactions

Although an autonomous transaction is started by another transaction, it is *not* a nested transaction:

- It does not share transactional resources (such as locks) with the main transaction.
- It does not depend on the main transaction. For example, if the main transaction rolls back, nested transactions roll back, but autonomous transactions do not.

- Its committed changes are visible to other transactions immediately. (A nested transaction's committed changes are not visible to other transactions until the main transaction commits.)
- Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.

Transaction Context

The main transaction shares its context with nested routines, but not with autonomous transactions. When one autonomous routine calls another (or itself recursively), the routines share no transaction context. When an autonomous routine calls a non-autonomous routine, the routines share the same transaction context.

Transaction Visibility

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. These changes become visible to the main transaction when it resumes, if its isolation level is set to `READ COMMITTED` (the default).

If you set the isolation level of the main transaction to `SERIALIZABLE`, changes made by its autonomous transactions are *not* visible to the main transaction when it resumes:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Controlling Autonomous Transactions

The first SQL statement in an autonomous routine begins a transaction. When one transaction ends, the next SQL statement begins another transaction. All SQL statements executed since the last commit or rollback make up the current transaction. To control autonomous transactions, use the following statements, which apply only to the current (active) transaction:

- `COMMIT`
- `ROLLBACK [TO savepoint_name]`
- `SAVEPOINT savepoint_name`
- `SET TRANSACTION`

Note: Transaction properties set in the main transaction apply only to that transaction, not to its autonomous transactions, and vice versa.

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. When one transaction ends, the next SQL statement begins another transaction. A single autonomous routine could contain several autonomous transactions, if it issued several `COMMIT` statements.

Using Savepoints

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. In an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

Avoiding Errors with Autonomous Transactions

To avoid some common errors, keep the following points in mind:

- If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur. Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.
- The Oracle initialization parameter `TRANSACTIONS` specifies the maximum number of concurrent transactions. That number might be exceeded because an autonomous transaction runs concurrently with the main transaction.
- If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception. If the exception goes unhandled, the transaction is rolled back.

Using Autonomous Triggers

Among other things, you can use database triggers to log events transparently. Suppose you want to track all inserts into a table, even those that roll back. In the example below, you use a trigger to insert duplicate rows into a shadow table. Because it is autonomous, the trigger can commit changes to the shadow table whether or not you commit changes to the main table.

```
-- 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 7, "Performing SQL Operations with Native Dynamic SQL"). 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 *Oracle Database 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 8-22.) To check for violations of the rules, you can use the pragma `RESTRICT_REFERENCES`. The pragma asserts that a function does not read or write database tables or package variables. (For more information, See *Oracle Database 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."

```
-- 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 BODYq 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;
   -- 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;
```

7

Performing SQL Operations with Native Dynamic SQL

A happy and gracious flexibility ... — Matthew Arnold

This chapter shows you how to use native dynamic SQL (dynamic SQL for short), a PL/SQL interface that makes your programs more flexible, by building and processing SQL statements at run time.

With dynamic SQL, you can directly execute any kind of SQL statement (even data definition and data control statements). You can build statements where you do not know table names, `WHERE` clauses, and other information in advance.

This chapter contains these topics:

What Is Dynamic SQL?

Some programs must build and process SQL statements where some information is not known in advance. A reporting application might build different `SELECT` statements for the various reports it generates, substituting new table and column names and ordering or grouping by different columns. Database management applications might issue statements such as `CREATE`, `DROP`, and `GRANT` that cannot be coded directly in a PL/SQL program. These statements are called *dynamic* SQL statements.

Dynamic SQL statements built as character strings built at run time. The strings contain the text of a SQL statement or PL/SQL block. They can also contain placeholders for bind arguments. Placeholder names are prefixed by a colon, and the names themselves do 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. To process a multi-row query (SELECT statement), you use the OPEN-FOR, FETCH, and CLOSE statements.

Why Use 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). Unlike INSERT, UPDATE, and DELETE statements, these statements cannot be included directly in a PL/SQL program.
- You want more flexibility. For example, you might want to pass the name of a schema object as a parameter to a procedure. You might want to build different search conditions for the WHERE clause of a SELECT statement.
- You want to issue a query where you do not know the number, names, or datatypes of the columns in advance. In this case, you use the DBMS_SQL package rather than the OPEN-FOR statement.

If you have older code that uses the DBMS_SQL package, the techniques described in this chapter using EXECUTE IMMEDIATE and OPEN-FOR generally provide better performance, more readable code, and extra features such as support for objects and collections. (For a comparison with DBMS_SQL, see *Oracle Database 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 main argument to EXECUTE IMMEDIATE is the string containing the SQL statement to execute. You can build up the string using concatenation, or use a predefined string.

Except for multi-row queries, the dynamic string can contain any SQL statement (without the final semicolon) or any PL/SQL block (with the final semicolon). The string can also contain placeholders, arbitrary names preceded by a colon, for bind arguments. In this case, you specify which PL/SQL variables correspond to the placeholders with the INTO, USING, and RETURNING INTO clauses.

You can only use placeholders in places where you can substitute variables in the SQL statement, such as conditional tests in WHERE clauses. You cannot use placeholders for the names of schema objects. For the right way, see "Passing Schema Object Names As Parameters" on page 7-9.

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. 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. 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 to Dynamic SQL" on page 7-10.

Dynamic SQL supports all the SQL datatypes. 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. For example, define variables and bind arguments cannot be Booleans or associative arrays. 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.

Example 7–1 Some Examples of Dynamic SQL

The following PL/SQL block contains several examples of dynamic SQL:

```
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;
/
```

Example 7–2 Dynamic SQL Procedure that Accepts Table Name and WHERE Clause

In the example below, a standalone procedure accepts the name of a database table and an optional WHERE-clause condition. If you omit the condition, the procedure deletes all rows from the table. Otherwise, the procedure deletes only those rows that meet the condition.

```
CREATE OR REPLACE 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;
END;
/
```

Specifying Parameter Modes for Bind Variables in Dynamic SQL Strings

With the `USING` clause, the mode defaults to `IN`, so you do not need to specify a parameter mode for input bind arguments.

With the `RETURNING INTO` clause, the mode is `OUT`, so you cannot specify a parameter mode for output bind arguments.

You must specify the parameter mode in more complicated cases, such as this one where you call a procedure from a dynamic PL/SQL block:

```
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;
/
```

Building a Dynamic Query with Dynamic SQL

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 (REF CURSORs)" on page 6-19.)

Examples of Dynamic SQL for Records, Objects, and Collections

Example 7–3 Dynamic SQL Fetching into a Record

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;
/
```

Example 7–4 Dynamic SQL for Object Types and Collections

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);
```

Using dynamic SQL, you can write a package that uses these types:

```
CREATE OR REPLACE 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 OR REPLACE 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 block, you might call the procedures in package TEAMS:

```
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

Bulk SQL passes entire collections back and forth, not just individual elements. This technique improves performance by minimizing the number of context switches between the PL/SQL and SQL engines. You can use a single statement instead of a loop that issues a SQL statement in every iteration.

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 DML Statements and Queries (FORALL, BULK COLLECT)" on page 11-7. Refer to that section for background information.

Using Dynamic SQL with Bulk SQL

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). 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.

EXECUTE IMMEDIATE

You can use the BULK COLLECT INTO clause with the EXECUTE IMMEDIATE statement to store values from each column of a query's result set in a separate collection.

You can use the RETURNING BULK COLLECT INTO clause with the EXECUTE IMMEDIATE statement to store the results of an INSERT, UPDATE, or DELETE statement in a set of collections.

FETCH

You can use the BULK COLLECT INTO clause with the FETCH statement to store values from each column of a cursor in a separate collection.

FORALL

You can put an EXECUTE IMMEDIATE statement with the RETURNING BULK COLLECT INTO inside a FORALL statement. You can store the results of all the INSERT, UPDATE, or DELETE statements in a set of collections.

You can pass subscripted collection elements to the EXECUTE IMMEDIATE statement through the USING clause. You cannot concatenate the subscripted elements directly into the string argument to EXECUTE IMMEDIATE; for example, you cannot build a collection of table names and write a FORALL statement where each iteration applies to a different table.

Examples of Dynamic Bulk Binds

Example 7–5 Dynamic SQL with BULK COLLECT INTO Clause

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:

```
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;
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;
/
```

Example 7–6 Dynamic SQL with RETURNING BULK COLLECT INTO Clause

Only INSERT, UPDATE, and DELETE statements can have output bind variables. You bulk-bind them with the RETURNING BULK COLLECT INTO clause of EXECUTE IMMEDIATE:

```
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;
/
```

Example 7–7 Dynamic SQL Inside FORALL Statement

To bind the input variables in a SQL statement, you can use the `FORALL` statement and `USING` clause, as shown below. 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;
/
```

Guidelines for Dynamic SQL

This section shows you how to take full advantage of dynamic SQL and how to avoid some common pitfalls.

When to Use or Omit the Semicolon with Dynamic SQL

When building up a single SQL statement in a string, do not include any semicolon at the end.

When building up a PL/SQL anonymous block, include the semicolon at the end of each PL/SQL statement and at the end of the anonymous block.

For example:

```
BEGIN
   EXECUTE IMMEDIATE 'dbms_output.put_line(''No semicolon'')';
   EXECUTE IMMEDIATE 'BEGIN dbms_output.put_line(''semicolons''); END;';
END;
```

Improving Performance of Dynamic SQL with Bind Variables

When you code `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements directly in PL/SQL, PL/SQL turns the variables into bind variables automatically, to make the

statements work efficiently with SQL. When you build up such statements in dynamic SQL, you need to specify the bind variables yourself to get the same 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 as each statement is parsed and cached.

```
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, which 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;
/
```

Passing Schema Object Names As Parameters

Suppose you need a procedure that accepts the name of any database table, then drops that table from your schema. You must build a string with a statement that includes the object names, then use `EXECUTE IMMEDIATE` to execute the statement:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE ' || table_name;
END;
/
```

Use concatenation to build the string, rather than trying to pass the table name as a bind variable through the `USING` clause.

Using Duplicate Placeholders with Dynamic SQL

Placeholders in a dynamic SQL statement are associated with bind arguments in the `USING` clause by position, not by name. If you specify a sequence of placeholders like `:a, :a, :b, :b`, you must include four items in the `USING` clause. For example, given the dynamic string

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

the fact that the name X is repeated is not significant. You can code the corresponding `USING` clause with four different bind variables:

```
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

If the dynamic statement represents a PL/SQL block, the rules for duplicate placeholders are different. Each unique placeholder maps to a single item in the `USING` clause. If the same placeholder appears two or more times, all references to that name correspond to one bind argument in the `USING` clause. In the following example, all references to the placeholder `X` are associated with the first bind argument `A`, and 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 with Dynamic SQL

The SQL cursor attributes %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT work when you issue an INSERT, UPDATE, DELETE, or single-row SELECT statement in dynamic SQL:

```
EXECUTE IMMEDIATE 'DELETE FROM employees WHERE employee_id > 1000';
rows_deleted := SQL%ROWCOUNT;
```

Likewise, when appended to a cursor variable name, the cursor attributes return information about the execution of a multi-row query:

```
OPEN c1 FOR 'SELECT * FROM employees';
FETCH c1 BULK COLLECT INTO rec_tab;
rows_fetched := c1%ROWCOUNT;
```

For more information about cursor attributes, see "Using Cursor Expressions" on page 6-27.

Passing Nulls to Dynamic SQL

The literal NULL is not allowed in the USING clause. To work around this restriction, replace the keyword NULL with an uninitialized variable:

```
DECLARE
   a_null CHAR(1); -- set to NULL automatically at run time
BEGIN
   EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING a_null;
END;
/
```

Using Database Links with Dynamic SQL

PL/SQL subprograms can execute dynamic SQL statements that use database links to refer to objects on remote databases:

```
PROCEDURE delete_dept (db_link VARCHAR2, dept_id INTEGER) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM departments@' || db_link ||
      ' WHERE deptno = :num' USING dept_id;
END;
/
```

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:

```
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:

```
DECLARE
   emp_count INTEGER;
BEGIN
   emp_count := row_count@chicago('employees');
END;
/
```

Using Invoker Rights with Dynamic SQL

Dynamic SQL lets you write schema-management procedures that can be centralized in one schema, and can be called from other schemas and operate on the objects in those schemas.

For example, this procedure can drop any kind of database object:

```
CREATE OR REPLACE PROCEDURE drop_it (kind IN VARCHAR2, name IN
VARCHAR2)
AUTHID CURRENT_USER
AS
BEGIN
   EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
END;
/
```

Let's say that this procedure is part of the `HR` schema. Without the `AUTHID` clause, the procedure would always drop objects in the `HR` schema, regardless of who calls it. Even if you pass a fully qualified object name, this procedure would not have the privileges to make changes in other schemas.

The `AUTHID` clause lifts both of these restrictions. It lets the procedure run with the privileges of the user that invokes it, and makes unqualified references refer to objects in that user's schema.

For details, see "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

Using Pragma RESTRICT_REFERENCES with Dynamic 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 8-22.) To check for violations of the rules, you can use the pragma `RESTRICT_REFERENCES`. The pragma asserts that a function does not read or write database tables or package variables. (For more information, See *Oracle Database Application Developer's Guide - Fundamentals*.)

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`). PL/SQL cannot detect those side-effects automatically, 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 with Dynamic SQL

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.

```
CREATE OR REPLACE PROCEDURE calc_bonus (emp_id NUMBER) AS
BEGIN
   EXECUTE IMMEDIATE 'DROP PROCEDURE calc_bonus'; -- deadlock!
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.

8

Using PL/SQL Subprograms

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 turn sets of statements into reusable subprograms. Subprograms are like building blocks for modular, maintainable applications.

This chapter contains these topics:

What Are Subprograms?

Subprograms are named PL/SQL blocks that can be called with a set of parameters. PL/SQL has two types of subprograms, *procedures* and *functions*. Generally, you use a procedure to perform an action and a function to compute a value.

Like anonymous blocks, subprograms have:

- A declarative part, with declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local and cease to exist when the subprogram ends.
- An executable part, with statements that assign values, control execution, and manipulate Oracle data.
- An optional exception-handling part, which deals with runtime error conditions.

Example 8–1 Simple PL/SQL Procedure

The following example shows a string-manipulation procedure that accepts both input and output parameters, and handles potential errors:

```
CREATE OR REPLACE PROCEDURE double
(
   original IN VARCHAR2, new_string OUT VARCHAR2
)
AS
BEGIN
   new_string := original || original;
   EXCEPTION
   WHEN VALUE_ERROR THEN
      dbms_output.put_line('Output buffer not long enough.');
END;
/
```

Example 8–2 Simple PL/SQL Function

The following example shows a numeric function that declares a local variable to hold temporary results, and returns a value when finished:

```
CREATE OR REPLACE FUNCTION square(original NUMBER)
RETURN NUMBER
AS
   original_squared NUMBER;
BEGIN
   original_squared := original * original;
   RETURN original_squared;
END;
/
```

Advantages of PL/SQL Subprograms

Subprograms let you extend the PL/SQL language. Procedures act like new statements. Functions act like new expressions and operators.

Subprograms let you break a program down into manageable, well-defined modules. You can use top-down design and the stepwise refinement approach to problem solving.

Subprograms promote reusability. Once tested, a subprogram can be reused in any number of applications. You can call PL/SQL subprograms from many different environments, so that you do not have to reinvent the wheel each time you use a new language or API to access the database.

Subprograms promote maintainability. You can change the internals of a subprogram without changing other subprograms that call it. Subprograms play a big part in other maintainability features, such as packages and object types.

Dummy subprograms (stubs) let you defer the definition of procedures and functions until after testing the main program. You can design applications from the top down, thinking abstractly, without worrying about implementation details.

When you use PL/SQL subprograms to define an API, you can make your code even more reusable and maintainable by grouping the subprograms into a PL/SQL package. For more information about packages, see Chapter 9, "Using PL/SQL Packages".

Understanding PL/SQL Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the SQL `CREATE PROCEDURE` statement. You specify the name of the procedure, its parameters, its local variables, and the `BEGIN-END` block that contains its code and handles any exceptions.

For each parameter, you specify:

- Its name.
- Its parameter mode (`IN`, `OUT`, or `IN OUT`). If you omit the mode, the default is `IN`. The optional `NOCOPY` keyword speeds up processing of large `OUT` or `IN OUT` parameters.
- Its datatype. You specify only the type, not any length or precision constraints.
- Optionally, its default value.

You can specify whether the procedure executes using the schema and permissions of the user who defined it, or the user who calls it. For more information, see "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

You can specify whether it should be part of the current transaction, or execute in its own transaction where it can `COMMIT` or `ROLLBACK` without ending the transaction of the caller. For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-35.

Procedures created this way are stored in the database. You can execute the `CREATE PROCEDURE` statement interactively from SQL*Plus, or from a program using native dynamic SQL (see Chapter 7, "Performing SQL Operations with Native Dynamic SQL").

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. The keyword `DECLARE` is used for anonymous PL/SQL blocks, but not procedures. 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. You can use the `NULL` statement to define a placeholder procedure or specify that the procedure does nothing. The exception-handling part contains exception handlers, which are placed between the keywords `EXCEPTION` and `END`.

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.

Functions have a number of optional keywords, used to declare a special class of functions known as table functions. They are typically used for transforming large amounts of data in data warehousing applications.

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 *Oracle Database 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.

A function is called as part of an expression:

```
IF sal_ok(new_sal, new_title) THEN ...
```

Using the RETURN Statement

The RETURN statement immediately ends the execution of a subprogram and returns control to the caller. Execution continues with the statement following the subprogram

call. (Do not confuse the RETURN statement with the RETURN clause in a function spec, which specifies the datatype of the return value.)

A subprogram can contain several RETURN statements. The subprogram does not have to conclude with a RETURN statement. Executing any RETURN statement completes the subprogram immediately.

In procedures, a RETURN statement does not return a value and so cannot contain an expression. The statement returns control to the caller before the end of the procedure.

In functions, a RETURN statement *must* contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable of the type specified in the RETURN clause. 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.

Declaring Nested PL/SQL Subprograms

You can declare subprograms in any PL/SQL block, subprogram, or package. The subprograms must go at the end of the declarative section, after all other items.

You must declare a subprogram before calling it. This requirement can make it difficult to declare several nested subprograms that call each other.

You can declare interrelated nested subprograms using a *forward declaration*: a subprogram spec terminated by a semicolon, with no body.

Although the formal parameter list appears in the forward declaration, it must also appear in the subprogram body. You can place the subprogram body anywhere after the forward declaration, but they must appear in the same program unit.

Example 8–3 Forward Declaration for a Nested Subprogram

```
DECLARE
   PROCEDURE proc1(arg_list);  -- forward declaration
   PROCEDURE proc2(arg_list);  -- calls proc1
   PROCEDURE proc1(arg_list) IS BEGIN proc2; END; -- calls proc2
BEGIN
```

```
   NULL;
END;
/
```

Passing Parameters to PL/SQL Subprograms

This section explains how to pass information in and out of PL/SQL subprograms using parameters:

Actual Versus Formal Subprogram Parameters

Subprograms pass information using *parameters*:

- The variables declared in a subprogram spec and referenced in the subprogram body are *formal* parameters.
- The variables or expressions passed from the calling subprogram are *actual* parameters.

A good programming practice is to use different names for actual and formal parameters.

When you call a procedure, 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, if you pass a number when the procedure expects a string, PL/SQL converts the parameter so that the procedure receives a string.

The actual parameter and its corresponding formal parameter must have compatible datatypes. For instance, PL/SQL cannot convert between the `DATE` and `REAL` datatypes, or convert a string to a number if the string contains extra characters such as dollar signs.

Example 8–4 Formal Parameters and Actual Parameters

The following procedure declares two formal parameters named `emp_id` and `amount`:

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
BEGIN
   UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
END raise_salary;
/
```

This procedure call specifies the actual parameters `emp_num` and `amount`:

```
raise_salary(emp_num, amount);
```

Expressions can be used as actual parameters:

```
raise_salary(emp_num, merit + cola);
```

Using Positional, Named, or Mixed Notation for Subprogram Parameters

When calling a subprogram, you can write the actual parameters using either:

- Positional notation. You specify the same parameters in the same order as they are declared in the procedure.

  This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. You must change your code if the procedure's parameter list changes.

- Named notation. You specify the name of each parameter along with its value. An arrow (=>) serves as the association operator. The order of the parameters is not significant.

  This notation is more verbose, but makes your code easier to read and maintain. You can sometimes avoid changing your code if the procedure's parameter list changes, for example if the parameters are reordered or a new optional parameter is added. Named notation is a good practice to use for any code that calls someone else's API, or defines an API for someone else to use.

- Mixed notation. You specify the first parameters with positional notation, then switch to named notation for the last parameters.

  You can use this notation to call procedures that have some required parameters, followed by some optional parameters.

Example 8–5 Subprogram Calls Using Positional, Named, and Mixed Notation

```
DECLARE
   acct INTEGER := 12345;
   amt  REAL := 500.00;
   PROCEDURE credit_acct (acct_no INTEGER, amount REAL) IS
      BEGIN NULL; END;
BEGIN
-- The following calls are all equivalent.
   credit_acct(acct, amt);                       -- positional
   credit_acct(amount => amt, acct_no => acct);  -- named
   credit_acct(acct_no => acct, amount => amt);  -- named
   credit_acct(acct, amount => amt);             -- mixed
END;
/
```

Specifying Subprogram Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes are `IN` (the default), `OUT`, and `IN OUT`.

Any parameter mode can be used with any subprogram. Avoid using the `OUT` and `IN OUT` modes with functions. To have a function return multiple values is a poor programming practice. Also, functions should be free from *side effects*, which change the values of variables not local to the subprogram.

Using the IN Mode

An `IN` parameter lets you pass values to the subprogram being called. Inside the subprogram, an `IN` parameter acts like a constant. It cannot be assigned a value.

You can pass a constant, literal, initialized variable, or expression as an IN parameter.

IN parameters can be initialized to default values, which are used if those parameters are omitted from the subprogram call. For more information, see "Using Default Values for Subprogram Parameters" on page 8-9.

Using the OUT Mode

An OUT parameter returns a value to the caller of a subprogram. Inside the subprogram, an OUT parameter acts like a variable. You can change its value, and reference the value after assigning it:

```
PROCEDURE split_name
(
   phrase IN VARCHAR2, first OUT VARCHAR2, last OUT VARCHAR2
)
IS
   first := SUBSTR(phrase, 1, INSTR(phrase, ' ')-1);
   last := SUBSTR(phrase, INSTR(phrase, ' ')+1);
   IF first = 'John' THEN
      DBMS_OUTPUT.PUT_LINE('That is a common first name.');
   END IF;
END;
/
```

You must pass a variable, not a constant or an expression, to an OUT parameter. Its previous value is lost unless you specify the NOCOPY keyword (see "Using Default Values for Subprogram Parameters" on page 8-9) or the subprogram exits with an unhandled exception.

Like variables, OUT formal parameters are initialized to NULL. The datatype of an OUT formal parameter cannot be a subtype defined as NOT NULL, such as the built-in subtypes NATURALN and POSITIVEN. Otherwise, when you call the subprogram, PL/SQL raises VALUE_ERROR.

Before exiting a subprogram, assign values to all OUT formal parameters. Otherwise, the corresponding actual parameters will be null. If you exit successfully, PL/SQL assigns values to the actual parameters. If you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

Using the IN OUT Mode

An IN OUT parameter passes initial values to a subprogram and returns updated values to the caller. It can be assigned a value and its value can be read. Typically, an IN OUT parameter is a string buffer or numeric accumulator, that is read inside the subprogram and then updated.

The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or an expression.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. If you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

Summary of Subprogram Parameter Modes

Table 8–1 summarizes all you need to know about the parameter modes.

Table 8–1 Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified

Table 8–1 (Cont.) Parameter Modes

IN	OUT	IN OUT
Passes values to a subprogram	Returns values to the caller	Passes initial values to a subprogram and returns updated values to the caller
Formal parameter acts like a constant	Formal parameter acts like an uninitialized variable	Formal parameter acts like an initialized variable
Formal parameter cannot be assigned a value	Formal parameter must be assigned a value	Formal parameter should be assigned a value
Actual parameter can be a constant, initialized variable, literal, or expression	Actual parameter must be a variable	Actual parameter must be a variable
Actual parameter is passed by reference (a pointer to the value is passed in)	Actual parameter is passed by value (a copy of the value is passed out) unless `NOCOPY` is specified	Actual parameter is passed by value (a copy of the value is passed in and out) unless `NOCOPY` is specified

Using Default Values for Subprogram Parameters

By initializing `IN` parameters to default values, you can pass different numbers of actual parameters to a subprogram, accepting the default values for any parameters you omit. You can also add new formal parameters without having to change every call to the subprogram.

Example 8–6 Procedure with Default Parameter Values

```
PROCEDURE create_dept (
    new_dname VARCHAR2 DEFAULT 'TEMP',
    new_loc   VARCHAR2 DEFAULT 'TEMP') IS
BEGIN
   NULL;
END;
/
```

If a parameter is omitted, the default value of its corresponding formal parameter is used. Consider the following calls to `create_dept`:

```
create_dept;             -- Same as create_dept('TEMP','TEMP');
create_dept('SALES');    -- Same as create_dept('SALES','TEMP');
create_dept('SALES', 'NY');
```

You cannot skip a formal parameter by leaving out its actual parameter. To omit the first parameter and specify the second, use named notation:

```
create_dept(new_loc => 'NEW YORK');
```

You cannot assign a null to an uninitialized formal parameter by leaving out its actual parameter. You must pass the null explicitly, or you can specify a default value of `NULL` in the declaration.

Overloading Subprogram Names

PL/SQL lets you **overload** subprogram names and type methods. You can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

Suppose you want to initialize the first *n* rows in two index-by tables that were declared as follows:

```
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
   NULL;
END;
/
```

You might write a procedure to initialize one kind of collection:

```
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
   END LOOP;
END initialize;
/
```

You might also write a procedure to initialize another kind of collection:

```
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, package, or object type. PL/SQL determines which procedure to call 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:

```
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
      NULL;
   END;
   PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
   BEGIN
      NULL;
   END;
BEGIN
   indx := 50;
   initialize(hiredate_tab, indx);  -- calls first version
   initialize(comm_tab, indx);      -- calls second version
END;
/
```

Guidelines for Overloading with Numeric Types

You can overload two subprograms if their formal parameters differ only in numeric datatype. This technique might be useful in writing mathematical APIs, where several versions of a function could use the same name, each accepting a different numeric type. For example, a function accepting `BINARY_FLOAT` might be faster, while a function accepting `BINARY_DOUBLE` might provide more precision.

To avoid problems or unexpected results passing parameters to such overloaded subprograms:

- Make sure to test that the expected version of a subprogram is called for each set of expected parameters. For example, if you have overloaded functions that accept `BINARY_FLOAT` and `BINARY_DOUBLE`, which is called if you pass a `VARCHAR2` literal such as '5.0'?
- Qualify numeric literals and use conversion functions to make clear what the intended parameter types are. For example, use literals such as `5.0f` (for `BINARY_FLOAT`), `5.0d` (for `BINARY_DOUBLE`), or conversion functions such as `TO_BINARY_FLOAT()`, `TO_BINARY_DOUBLE()`, and `TO_NUMBER()`.

PL/SQL looks for matching numeric parameters starting with `PLS_INTEGER` or `BINARY_INTEGER`, then `NUMBER`, then `BINARY_FLOAT`, then `BINARY_DOUBLE`. The first overloaded subprogram that matches the supplied parameters is used. A `VARCHAR2` value can match a `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE` parameter.

For example, consider the `SQRT` function, which takes a single parameter. There are overloaded versions that accept a `NUMBER`, a `BINARY_FLOAT`, or a `BINARY_DOUBLE` parameter. If you pass a `PLS_INTEGER` parameter, the first matching overload (using the order given in the preceding paragraph) is the one with a `NUMBER` parameter, which is likely to be the slowest. To use one of the faster versions, use the `TO_BINARY_FLOAT` or `TO_BINARY_DOUBLE` functions to convert the parameter to the right datatype.

For another example, consider the `ATAN2` function, which takes two parameters of the same type. If you pass two parameters of the same type, you can predict which overloaded version is used through the same rules as before. If you pass parameters of different types, for example one `PLS_INTEGER` and one `BINARY_FLOAT`, PL/SQL tries to find a match where both parameters use the "higher" type. In this case, that is the version of `ATAN2` that takes two `BINARY_FLOAT` parameters; the `PLS_INTEGER` parameter is converted "upwards".

The preference for converting "upwards" holds in more complicated situations. For example, you might have a complex function that takes two parameters of different types. One overloaded version might take a `PLS_INTEGER` and a `BINARY_FLOAT` parameter. Another overloaded version might take a `NUMBER` and a `BINARY_DOUBLE` parameter. What happens if you call this procedure name and pass two `NUMBER` parameters? PL/SQL looks "upward" first to find the overloaded version where the second parameter is `BINARY_FLOAT`. Because this parameter is a closer match than the `BINARY_DOUBLE` parameter in the other overload, PL/SQL then looks "downward" and converts the first `NUMBER` parameter to `PLS_INTEGER`.

Restrictions on Overloading

Only local or packaged subprograms, or type methods, can be overloaded. You cannot overload standalone subprograms.

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:

```
DECLARE
   PROCEDURE reconcile (acct_no IN INTEGER) IS
   BEGIN NULL; END;
   PROCEDURE reconcile (acct_no OUT INTEGER) IS
   BEGIN NULL; END;
/
```

You cannot overload subprograms whose parameters differ only in subtype. For example, you cannot overload procedures where one accepts an `INTEGER` parameter and the other accepts a `REAL` parameter, even though `INTEGER` and `REAL` are both subtypes of `NUMBER` and so are in the same family.

You cannot overload two functions that differ only in the datatype of the return value, even if the types are in different families. For example, you cannot overload two functions where one returns `BOOLEAN` and the other returns `INTEGER`.

How Subprogram Calls Are Resolved

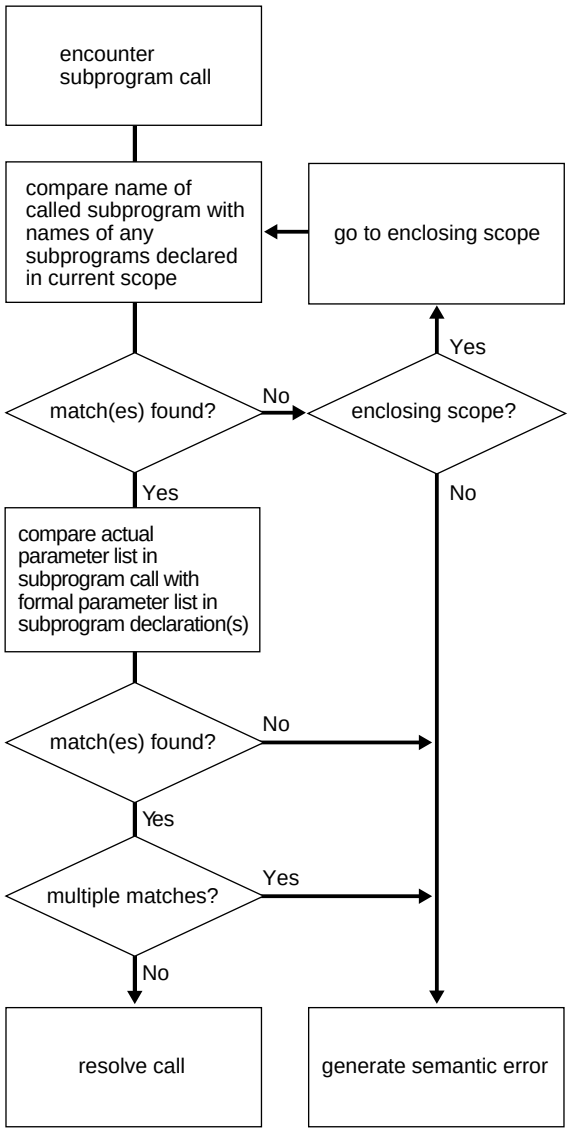
Figure 8–1 shows how the PL/SQL compiler resolves subprogram calls. When the compiler encounters a procedure or function call, it tries to find a declaration that matches the call. The compiler searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler looks more closely when it finds one or more subprogram declarations in which the subprogram name matches the name of the called subprogram.

To resolve a call among possibly like-named subprograms at the same level of scope, the compiler must find an *exact* match between the actual and formal parameters. They must match in number, order, and datatype (unless some formal parameters were assigned default values). If no match is found or if multiple matches are found, the compiler generates a semantic error.

The following example calls the enclosing procedure `swap` from the function `reconcile`, generating an error because neither declaration of `swap` within the current scope matches the procedure call:

```
PROCEDURE swap (n1 NUMBER, n2 NUMBER) IS
   num1 NUMBER;
   num2 NUMBER;
   FUNCTION balance (...) RETURN REAL IS
    PROCEDURE swap (d1 DATE, d2 DATE) IS BEGIN NULL; END;
    PROCEDURE swap (b1 BOOLEAN, b2 BOOLEAN) IS BEGIN NULL; END;
   BEGIN
      swap(num1, num2);
      RETURN ...
   END balance;
BEGIN NULL; END;
/
```

Figure 8–1 How the PL/SQL Compiler Resolves Calls



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:

```
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:

```
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.

```
set serveroutput on
declare
v final_t := final_t(1,2,3);
begin
  dbms_output.put_line(p.foo(v));
end;
/
```

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.

```
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**.

```
set serveroutput on
declare
  v super_t2 := final_t2(1,2,3);
begin
  dbms_output.put_line(v.foo);
end;
/
```

Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)

By default, stored procedures and SQL methods execute with the privileges of their owner, not their current user. Such *definer's rights* subprograms are bound to the schema in which they reside, allowing you to refer to objects in the same schema without qualifying their names. For example, if schemas SCOTT and BLAKE both have a table called dept, a procedure owned by SCOTT can refer to dept rather than SCOTT.DEPT. If user BLAKE calls SCOTT's procedure, the procedure still accesses the dept table owned by SCOTT.

If you compile the same procedure in both schemas, you can define the schema name as a variable in SQL*Plus and refer to the table like &schema..dept. The code is portable, but if you change it, you must recompile it in each schema.

A more maintainable way is to use the AUTHID clause, which makes stored procedures and SQL methods execute with the privileges and schema context of the calling user. You can create one instance of the procedure, and many users can call it to access their own data.

Such *invoker's rights* subprograms are not bound to a particular schema. The following version of procedure create_dept executes with the privileges of the calling user and inserts rows into that user's dept table:

```
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's Rights

Invoker's rights subprograms let you reuse code and centralize application logic. They are especially useful in applications that store data using identical tables in different schemas. All the schemas in one instance can call procedures owned by a central schema. You can even have schemas in different instances call centralized procedures using a database link.

Consider a company that uses a stored procedure to analyze sales. If the company has several schemas, each with a similar SALES table, normally it would also need several copies of the stored procedure, one in each schema.

To solve the problem, the company installs an invoker's rights version of the stored procedure in a central schema. Now, all the other schemas can call the same procedure, which queries the appropriate to SALES table in each case.

You can restrict access to sensitive data by calling from an invoker's rights subprogram to a definer's rights subprogram that queries or updates the table containing the sensitive data. Although multiple users can call the invoker's rights subprogram, they do not have direct access to the sensitive data.

Specifying the Privileges for a Subprogram with the AUTHID Clause

To implement invoker's 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. In the CREATE FUNCTION, CREATE PROCEDURE, CREATE PACKAGE, or CREATE TYPE statement, you can include either AUTHID CURRENT_USER or AUTHID DEFINER immediately before the IS or AS keyword that begins the declaration section.

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's rights packages.

Who Is the Current User During Subprogram Execution?

In a sequence of calls, whenever control is inside an invoker's rights subprogram, the current user is the session user. When a definer's rights subprogram is called, the owner of that subprogram becomes the current user. The current user might change as new subprograms are called or as subprograms exit.

To verify who the current user is at any time, you can check the USER_USERS data dictionary view. Inside an invoker's rights subprogram, the value from this view might be different from the value of the USER built-in function, which always returns the name of the session user.

How External References Are Resolved in Invoker's 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.

```
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's Rights Subprograms

The PL/SQL compiler must resolve all references to tables and other objects at compile time. The owner of an invoker's rights subprogram must have objects in the same schema with the right names and columns, even if they do not contain any data. At run time, the corresponding objects in the caller's schema must have matching definitions. Otherwise, you get an error or unexpected results, such as ignoring table columns that exist in the caller's schema but not in the schema that contains the subprogram.

Overriding Default Name Resolution in Invoker's Rights Subprograms

Occasionally, you might want an unqualified name to refer to some particular schema, not the schema of the caller. In the same schema as the invoker's rights subprogram, create a public synonym for the table, procedure, function, or other object using the CREATE SYNONYM statement:

```
CREATE PUBLIC SYNONYM emp FOR hr.employees;
```

When the invoker's rights subprogram refers to this name, it will match the synonym in its own schema, which resolves to the object in the specified schema. This technique does not work if the calling schema already has a schema object or private synonym with the same name. In that case, the invoker's rights subprogram must fully qualify the reference.

Granting Privileges on Invoker's 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's rights subprogram operates under the security domain of its owner, no matter who is executing it. The owner must have the privileges needed to access schema objects referenced by the subprogram.

You can write a program consisting of multiple subprograms, some with definer's rights and others with invoker's rights. Then, you can use the EXECUTE privilege to restrict program entry points. That way, users of an entry-point subprogram can execute the other subprograms indirectly but not directly.

Granting Privileges on an Invoker's Rights Subprogram: Example

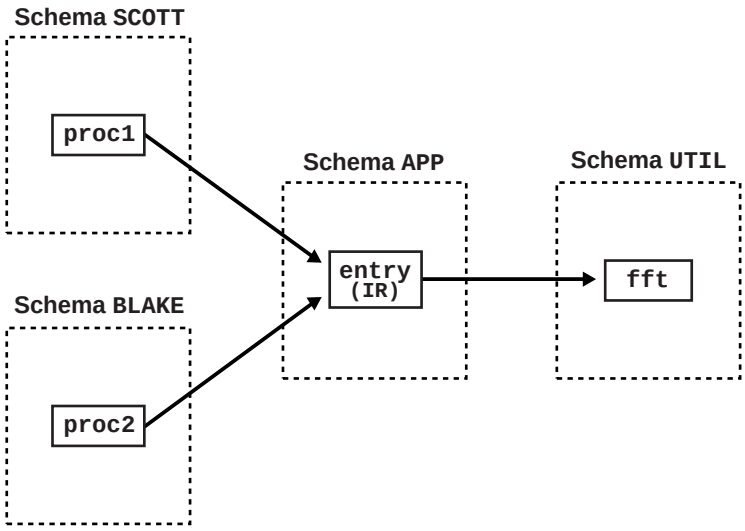
Suppose user UTIL grants the EXECUTE privilege on subprogram FFT to user APP:

```
GRANT EXECUTE ON util.fft TO app;
```

Now, user APP can compile functions and procedures that call subprogram FFT. At run time, no privilege checks on the calls are done. As Figure 8–2 shows, user UTIL need not grant the EXECUTE privilege to every user who might call FFT indirectly.

Since subprogram `util.fft` is called directly only from invoker's rights subprogram `app.entry`, 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.

Figure 8–2 Indirect Calls to an Invoker's Rights Subprogram



Using Roles with Invoker's Rights Subprograms

The use of roles in a subprogram depends on whether it executes with definer's rights or invoker's rights. Within a definer's rights subprogram, all roles are disabled. Roles are not used for privilege checking, and you cannot set roles.

Within an invoker's rights subprogram, roles are enabled (unless the subprogram was called directly or indirectly by a definer's 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's Rights Subprograms

For invoker's rights subprograms executed within a view expression, the schema that created the view, not the schema that is querying the view, is considered to be the current user.

This rule also applies to database triggers.

Using Database Links with Invoker's Rights Subprograms

You can create a database link to use invoker's rights:

```
CREATE DATABASE LINK link_name CONNECT TO CURRENT_USER
   USING connect_string;
```

A current-user link lets you connect to a remote database as another user, with that user's privileges. To connect, Oracle uses the username of the current user (who must be a global user). Suppose an invoker's 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.

```
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER USING ...
```

If it were a definer's rights subprogram, the current user would be BLAKE, and the subprogram would connect to the Dallas database as global user BLAKE.

Using Object Types with Invoker's 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 12, "Using PL/SQL Object Types".) Suppose user BLAKE creates the following object type:

```
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:

```
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:

```
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's Rights Instance Methods

An invoker's rights instance method executes with the privileges of the invoker, not the creator of the instance. Suppose that Person is an invoker's 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:

```
-- 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;
/
```

Using Recursion with PL/SQL

Recursion is a powerful technique for simplifying the design of algorithms. Basically, *recursion* means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ...), is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined as simpler versions of itself. Consider the definition of *n* factorial (*n!*), the product of all integers from 1 to *n*:

```
n! = n * (n - 1)!
```

What Is a Recursive Subprogram?

A recursive subprogram is one that calls itself. Each recursive call creates a new instance of any items declared in the subprogram, including parameters, variables, cursors, and exceptions. Likewise, new instances of SQL statements are created at each level in the recursive descent.

Be careful where you place a recursive call. If you place it inside a cursor FOR loop or between OPEN and CLOSE statements, another cursor is opened at each call, which might exceed the limit set by the Oracle initialization parameter OPEN_CURSORS.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a terminating condition. Otherwise, the recursion would go on until PL/SQL runs out of memory and raises the predefined exception `STORAGE_ERROR`.

Calling External Subprograms

Although PL/SQL is a powerful, flexible language, some tasks are more easily done in another language. Low-level languages such as C are very fast. Widely used languages such as Java have reusable libraries for common design patterns.

You can use PL/SQL call specs to invoke *external subprograms* written in other languages, making their capabilities and libraries available 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:

```
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:

```
CREATE PROCEDURE raise_salary (empno NUMBER, pct NUMBER)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
```

You might call procedure `raise_salary` from an anonymous PL/SQL block:

```
DECLARE
   emp_id  NUMBER;
   percent NUMBER;
BEGIN
   -- get values for emp_id and percent
   raise_salary(emp_id, percent);  -- call external subprogram
END;
/
```

External C subprograms are used to interface with embedded systems, solve engineering problems, analyze data, or control real-time devices and processes. External C subprograms extend the functionality of the database server, and move computation-bound programs from client to server, where they execute faster.

For more information about Java stored procedures, see *Oracle Database Java Developer's Guide*. For more information about external C subprograms, see *Oracle Database 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 *Oracle Database Application Developer's Guide - Fundamentals*.

Controlling Side Effects of PL/SQL Subprograms

To be callable from SQL statements, a stored function (and any subprograms called by that function) must obey certain "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 or write database tables 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`.

For full syntax details, see "RESTRICT_REFERENCES Pragma" on page 13-113. For more information about the purity rules, see *Oracle Database Application Developer's Guide - Fundamentals*.

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; 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.

Example 8–7 Aliasing from Passing Global Variable with NOCOPY Hint

In the example below, procedure `ADD_ENTRY` refers to varray `LEXICON` both as a parameter and as a global variable. When `ADD_ENTRY` is called, the identifiers `WORD_LIST` and `LEXICON` point to 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);
END;
/
```

The program prints `aardwolf` if the compiler obeys the `NOCOPY` hint. The assignment to `WORD_LIST` is done immediately through a pointer, then is overwritten by the assignment to `LEXICON`.

The program prints `aardvark` if the `NOCOPY` hint is omitted, or if the compiler does not obey the hint. The assignment to `WORD_LIST` uses an internal copy of the varray, which is copied back to the actual parameter (overwriting the contents of `LEXICON`) when the procedure ends.

Example 8–8 Aliasing Passing Same Parameter Multiple Times

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.

```
 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;
/
```

Example 8–9 Aliasing from Assigning Cursor Variables to Same Work Area

Because they are pointers, cursor variables also increase the possibility of aliasing. In the following example, after the assignment, `emp_cv2` is an alias of `emp_cv1`; both point to the same query work area. The first fetch from `emp_cv2` fetches the third row, not the first, because the first two rows were already fetched from `emp_cv1`. The second fetch from `emp_cv2` fails because `emp_cv1` is closed.

```
PROCEDURE get_emp_data (
   emp_cv1 IN OUT EmpCurTyp,
   emp_cv2 IN OUT EmpCurTyp) IS
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_cv1 FOR SELECT * FROM employees;
   emp_cv2 := emp_cv1;
   FETCH emp_cv1 INTO emp_rec;  -- fetches first row
   FETCH emp_cv1 INTO emp_rec;  -- fetches second row
   FETCH emp_cv2 INTO emp_rec;  -- fetches third row
   CLOSE emp_cv1;
   FETCH emp_cv2 INTO emp_rec;  -- raises INVALID_CURSOR
END;
/
```

9

Using PL/SQL Packages

Goods which are not shared are not goods. —Fernando de Rojas

This chapter shows how to bundle related PL/SQL code and data into a package. The package might include a set of procedures that forms an API, or a pool of type definitions and variable declarations. The package is compiled and stored in the database, where its contents can be shared by many applications.

This chapter contains these topics:

What Is a PL/SQL Package?

A **package** is a schema object that groups logically related PL/SQL types, variables, and subprograms. Packages usually have two parts, a specification and a body; sometimes the body is unnecessary. The **specification** (**spec** for short) is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. The **body** defines the queries for the cursors and the code for the subprograms.

You can think of the spec as an interface and of the body as a "black box." You can debug, enhance, or replace a package body without changing the package spec.

To create package specs, use the SQL statement `CREATE PACKAGE`. If necessary, a `CREATE PACKAGE BODY` statement defines the package body.

The spec holds **public declarations**, which are visible to stored procedures and other code outside the package. You must declare subprograms at the end of the spec after all other items (except pragmas that name a specific function; such pragmas must follow the function spec).

The body holds implementation details and **private declarations**, which are hidden from code outside the package. Following the declarative part of the package body is the optional initialization part, which holds statements that initialize package variables and do any other one-time setup steps.

The `AUTHID` clause determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

A **call spec** lets you map a package subprogram to a Java method or external C function. The call spec maps the Java or C name, parameter types, and return type to their SQL counterparts. To learn how to write Java call specs, see *Oracle Database Java Developer's Guide*. To learn how to write C call specs, see *Oracle Database Application Developer's Guide - Fundamentals*.

What Goes In a PL/SQL Package?

- "Get" and "Set" methods for the package variables, if you want to avoid letting other procedures read and write them directly.
- Cursor declarations with the text of SQL queries. Reusing exactly the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if you need to change a query that is used in many places.
- Declarations for exceptions. Typically, you need to be able to reference these from different procedures, so that you can handle exceptions within called subprograms.
- Declarations for procedures and functions that call each other. You do not need to worry about compilation order for packaged procedures and functions, making them more convenient than standalone stored procedures and functions when they call back and forth to each other.
- Declarations for overloaded procedures and functions. You can create multiple variations of a procedure or function, using the same names but different sets of parameters.

- Variables that you want to remain available between procedure calls in the same session. You can treat variables in a package like global variables.
- Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored procedures or functions, you must declare the type in a package so that both the calling and called subprogram can refer to it.

Example of a PL/SQL Package

The example below packages a record type, a cursor, and two employment procedures. The procedure `hire_employee` uses the sequence `empno_seq` and the function `SYSDATE` to insert a new employee number and hire date.

```
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. You can change the body (implementation) without having to recompile calling programs.

Advantages of PL/SQL Packages

Packages have a long history in software engineering, offering important features for reliable, maintainable, reusable code, often in team development efforts for large systems.

Modularity

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.

Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

Added Functionality

Packaged public variables and cursors persist for the duration of a session. They can be shared by all subprograms that execute in the environment. They let you maintain data across transactions without storing it in the database.

Better Performance

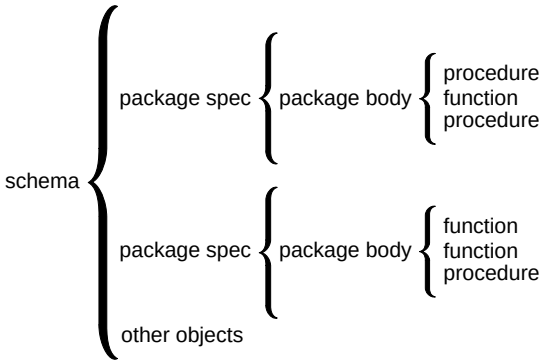
When you call a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O.

Packages stop cascading dependencies and avoid unnecessary recompiling. For example, if you change the body of a packaged function, Oracle does not recompile other subprograms that call the function; these subprograms only depend on the parameters and return value that are declared in the spec, so they are only recompiled if the spec changes.

Understanding The Package Specification

The package specification contains public declarations. The declared items are accessible from anywhere in the package and to any other subprograms in the same schema. Figure 9–1 illustrates the scoping.

Figure 9–1 Package Scope



The spec lists the package resources available to applications. All the information your application needs to use the resources is in the spec. For example, the following declaration shows that the function named `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).

If a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. Only subprograms and cursors have an underlying implementation. In the following example, the package needs no body because it declares types, exceptions, and variables, but no subprograms or cursors. Such packages let you define global variables—usable by stored procedures and functions and triggers—that persist throughout a session.

```
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);
   minimum_balance    CONSTANT REAL := 10.00;
   number_processed   INT;
   insufficient_funds EXCEPTION;
END trans_data;
/
```

Referencing Package Contents

To reference the types, items, subprograms, and call specs declared within a package spec, use dot notation:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
package_name.call_spec_name
```

You can reference package contents from database triggers, stored subprograms, 3GL application programs, and various Oracle tools. For example, you might call the packaged procedure `hire_employee` from SQL*Plus, as follows:

```
CALL emp_actions.hire_employee('TATE', 'CLERK', ...);
```

The following example calls the same procedure from an anonymous block in a Pro*C program. The actual parameters `emp_name` and `job_title` are host variables.

```
EXEC SQL EXECUTE
   BEGIN
      emp_actions.hire_employee(:emp_name, :job_title, ...);
```

Restrictions

You cannot reference remote packaged variables, either directly or indirectly. For example, you cannot call the a procedure through a database link if the procedure refers to a packaged variable.

Inside a package, you cannot reference host variables.

Understanding The Package Body

The package body contains the implementation of every cursor and subprogram declared in the package spec. Subprograms defined in a package body are accessible outside the package only if their specs also appear in the package spec. If a subprogram spec is not included in the package spec, that subprogram can only be called by other subprograms in the same package.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. Except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception, as the following example shows:

```
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 following package, 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 use by any application that has execute privilege on the package.

```
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;

   /* Fully define subprograms specified in package. */
   FUNCTION hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    REAL,
      sal    REAL,
```

```
      comm   REAL,
      deptno 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. In the last example, only one row is inserted into the database table `emp_audit`, and the variable `number_hired` is initialized only once.

Every time the procedure `hire_employee` is 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.

The following example is a package that handles 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);
END bank_transactions;
/

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;
```

```
/* Define local procedures, available only in package. */
PROCEDURE do_journal_entry (
/* Record transaction in journal. */
   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
/* Credit account unless account number is bad. */
   old_balance REAL;
   new_balance REAL;
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
/* Debit account unless account number is bad or
   account has insufficient funds. */
   old_balance REAL;
   new_balance REAL;
   insufficient_funds EXCEPTION;
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;
   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

In the package `emp_actions`, the package body declares a variable named `number_hired`, which is initialized to zero. Items declared in the body are restricted to use within the package. PL/SQL code outside the package cannot reference the variable `number_hired`. Such items are called **private**.

Items declared in the spec of `emp_actions`, such as the exception `invalid_salary`, are visible outside the package. Any PL/SQL code can reference the exception `invalid_salary`. Such items are called **public**.

To maintain items throughout a session or across transactions, place them in the declarative part of the package body. For example, the value of `number_hired` is kept between calls to `hire_employee` within the same session. The value is lost when the session ends.

To make the items public, place them in the package 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 8-9.

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:

```
abs_diff := ABS(x - y);
```

If you declare your own version of ABS, your local declaration overrides the global declaration. You can still call the built-in function by specifying its full name:

```
abs_diff := STANDARD.ABS(x - y);
```

Most built-in functions are overloaded. For example, package STANDARD contains the following declarations:

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves a call to TO_CHAR by matching the number and datatypes of the formal and actual parameters.

Overview of Product-Specific Packages

Oracle and various Oracle tools are supplied with product-specific packages that define APIs you can call from PL/SQL, SQL, Java, or other programming environments. Here we mention a few of the more widely used ones. For more information, see *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:

```
CREATE OR REPLACE PROCEDURE list_tables AS
BEGIN
   dbms_output.put_line('These are the tables you own:');
   FOR item IN (SELECT table_name FROM user_tables)
   LOOP
      dbms_output.put_line(item.table_name);
   END LOOP;
END;
/
```

When you issue the following commands, SQL*Plus displays the output from the procedure:

```
SQL> SET SERVEROUTPUT ON
SQL> EXEC list_tables;
```

If the output is long, you might need to issue `SET SERVEROUTPUT ON SIZE 1000000` to use a bigger output buffer.

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 or to a waiting application such as a UNIX program.

At the other end of the pipe, you can use the procedures `receive_message` and `unpack_message` to receive and unpack (read) the message. Named pipes are useful in many ways. For example, you can write a C program to collect data, then send it through pipes to stored procedures in an Oracle database.

About the UTL_FILE Package

Package `UTL_FILE` lets PL/SQL programs 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 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 general so they can be reused in future applications. Become familiar with the Oracle-supplied packages, and avoid writing packages that duplicate features already provided by Oracle.

Design and define package specs before the package bodies. Place in a spec only those things that must be visible to calling programs. That way, other developers cannot build unsafe dependencies on your implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package spec. Changes to a package body do not require recompiling calling procedures. Changes to a package spec require Oracle to recompile every stored subprogram that references the package.

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 example, you can change the cursor body in the last example, without having to change the cursor spec.

From a PL/SQL block or subprogram, you use dot notation to reference a packaged cursor, as the following example shows:

```
DECLARE
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_stuff.c1;
   LOOP
      FETCH emp_stuff.c1 INTO emp_rec; /* Do more processing here... */
      EXIT WHEN emp_suff.c1%NOTFOUND;
   END LOOP;
   CLOSE emp_stuff.c1;
END;
/
```

The scope of a packaged cursor is not limited to a PL/SQL block. When you open a packaged cursor, it remains open until you close it or you disconnect from the session.

10

Handling PL/SQL Errors

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 contains these topics:

Overview of PL/SQL Runtime Error Handling

In PL/SQL, an error condition is called an *exception*. Exceptions can be internally defined (by the runtime 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.

The following example calculates a price-to-earnings ratio for a company. If the company has zero earnings, the division operation raises the predefined exception ZERO_DIVIDE, the execution of the block is interrupted, and control is transferred to the exception handlers. The optional OTHERS handler catches all exceptions that the block does not name specifically.

```
SET SERVEROUTPUT ON;

DECLARE
   stock_price NUMBER := 9.73;
   net_earnings NUMBER := 0;
   pe_ratio NUMBER;
BEGIN
-- Calculation might cause division-by-zero error.
   pe_ratio := stock_price / net_earnings;
   dbms_output.put_line('Price/earnings ratio = ' || pe_ratio);

EXCEPTION  -- exception handlers begin

-- Only one of the WHEN blocks is executed.

   WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
      dbms_output.put_line('Company must have had zero earnings.');
      pe_ratio := null;

   WHEN OTHERS THEN  -- handles all other errors
      dbms_output.put_line('Some other kind of error occurred.');
      pe_ratio := null;

END;  -- exception handlers and block end here
/
```

The last example illustrates exception handling. With some better error checking, we could have avoided the exception entirely, by substituting a null for the answer if the denominator was zero:

```
DECLARE
   stock_price NUMBER := 9.73;
   net_earnings NUMBER := 0;
   pe_ratio NUMBER;
BEGIN
   pe_ratio :=
      case net_earnings
         when 0 then null
         else stock_price / net_earnings
      end;
END;
/
```

Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

- Add exception handlers whenever there is any possibility of an error occurring. Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors could also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still needs to take corrective action.
- Add error-checking code whenever you can predict that an error might occur if your code gets bad input data. Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.
- Make your programs robust enough to work even if the database is not in the state you expect. For example, perhaps a table you query will have columns added or deleted, or their types changed. You can avoid such problems by declaring individual variables with `%TYPE` qualifiers, and declaring records to hold query results with `%ROWTYPE` qualifiers.
- Handle named exceptions whenever possible, instead of using WHEN OTHERS in exception handlers. Learn the names and causes of the predefined exceptions. If your database operations might cause particular ORA- errors, associate names with these errors so you can write handlers for them. (You will learn how to do that later in this chapter.)
- Test your code with different combinations of bad data to see what potential errors arise.
- Write out debugging information in your exception handlers. You might store such information in a separate table. If so, do it by making a call to a procedure declared with the `PRAGMA AUTONOMOUS_TRANSACTION`, so that you can commit your debugging information, even if you roll back the work that the main procedure was doing.
- Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue. Remember, no matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.

Advantages of PL/SQL Exceptions

Using exceptions for error handling has several advantages.

With exceptions, you can reliably handle potential errors from many statements with a single exception handler:

```
BEGIN
   SELECT ...
   SELECT ...
   procedure_that_performs_select();
   ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
```

Instead of checking for an error at every point it might occur, just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.

Sometimes the error is not immediately obvious, and could not be detected until later when you perform calculations using bad data. Again, a single exception handler can trap all division-by-zero errors, bad array subscripts, and so on.

If you need to check for errors at a specific spot, you can enclose a single statement or a group of statements inside its own BEGIN-END block with its own exception handler. You can make the checking as general or as precise as you like.

Isolating error-handling routines makes the rest of the program easier to read and understand.

Summary of Predefined PL/SQL Exceptions

An internal exception is raised automatically if your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

You can use the pragma `EXCEPTION_INIT` to associate exception names with other Oracle error codes that you can anticipate. To handle unexpected Oracle errors, you can use the `OTHERS` handler. Within this handler, you can call the functions `SQLCODE` and `SQLERRM` to return the Oracle error code and message text. Once you know the error code, you can use it with pragma `EXCEPTION_INIT` and write a handler specifically for that error.

PL/SQL declares predefined exceptions globally in package `STANDARD`. You need not declare them yourself. You can write handlers for predefined exceptions using the names in the following list:

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51

Exception	Oracle Error	SQLCODE Value
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Brief descriptions of the predefined exceptions follow:

Exception	Raised when ...
ACCESS_INTO_NULL	A program attempts to assign values to the attributes of an uninitialized object.
CASE_NOT_FOUND	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	A program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	A 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.
DUP_VAL_ON_INDEX	A program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	A program attempts a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	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.
LOGIN_DENIED	A program attempts to log on to Oracle with an invalid username or password.
NO_DATA_FOUND	A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. Because this exception is used internally by some SQL functions to signal that they are finished, you should not rely on this exception being propagated if you raise it within a function that is called as part of a query.
NOT_LOGGED_ON	A program issues a database call without being connected to Oracle.
PROGRAM_ERROR	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	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.
SELF_IS_NULL	A program attempts to call a MEMBER method, but the instance of the object type has not been initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method.

Exception	Raised when ...
STORAGE_ERROR	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	A program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
VALUE_ERROR	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.)
ZERO_DIVIDE	A program attempts to divide a number by zero.

Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements.

Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION. In the following example, you declare an exception named past_due:

```
DECLARE
   past_due EXCEPTION;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules for PL/SQL Exceptions

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. The sub-block cannot reference the global exception, unless the exception is declared in a labeled block and you qualify its name with the block label:

```
block_label.exception_name
```

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;
     due_date DATE := SYSDATE - 1;
     todays_date DATE := SYSDATE;
   BEGIN
      IF due_date < todays_date THEN
         RAISE past_due;  -- this is not handled
      END IF;
   END;  ------------- sub-block ends
EXCEPTION
   WHEN past_due THEN  -- does not handle RAISEd exception
      dbms_output.put_line('Handling PAST_DUE exception.');
   WHEN OTHERS THEN
     dbms_output.put_line('Could not recognize PAST_DUE_EXCEPTION in this
scope.');
END;
/
```

The enclosing block does not handle the raised exception because the declaration of `past_due` in the sub-block prevails. Though they share the same name, the two `past_due` exceptions are different, just as the two `acct_num` variables share the same name but are different variables. Thus, the `RAISE` statement and the `WHEN` clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the sub-block or define an `OTHERS` handler.

Associating a PL/SQL Exception with a Number: Pragma EXCEPTION_INIT

To handle error conditions (typically `ORA-` messages) that have no predefined name, you must use the `OTHERS` handler or the pragma `EXCEPTION_INIT`. A **pragma** is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma `EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an **error stack**, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma `EXCEPTION_INIT` in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

where `exception_name` is the name of a previously declared exception and the number is a negative value corresponding to an `ORA-` error number. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:

```
DECLARE
   deadlock_detected EXCEPTION;
```

```
   PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
   null; -- Some operation that causes an ORA-00060 error
EXCEPTION
   WHEN deadlock_detected THEN
      null; -- handle the error
END;
/
```

Defining Your Own Error Messages: Procedure RAISE_APPLICATION_ERROR

The procedure RAISE_APPLICATION_ERROR lets you issue user-defined ORA- error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To 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. RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD, and as with package STANDARD, you do not need to qualify references to it.

An application can 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 error condition of your choosing happens (in this case, if the current schema owns less than 1000 tables):

```
DECLARE
   num_tables NUMBER;
BEGIN
   SELECT COUNT(*) INTO num_tables FROM USER_TABLES;
   IF num_tables < 1000 THEN
      /* Issue your own error code (ORA-20101) with your own error message. */
      raise_application_error(-20101, 'Expecting at least 1000 tables');
   ELSE
      NULL; -- Do the rest of the processing (for the non-error case).
   END IF;
END;
/
```

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
END;
```

How PL/SQL Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle error number using `EXCEPTION_INIT`. However, other user-defined exceptions must be raised explicitly by `RAISE` statements.

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`:

```
DECLARE
   out_of_stock   EXCEPTION;
   number_on_hand NUMBER := 0;
BEGIN
   IF number_on_hand < 1 THEN
      RAISE out_of_stock; -- raise an exception that we defined
   END IF;
EXCEPTION
   WHEN out_of_stock THEN
      -- handle the error
      dbms_output.put_line('Encountered out-of-stock error.');
END;
/
```

You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as the following example shows:

```
DECLARE
   acct_type INTEGER := 7;
BEGIN
   IF acct_type NOT IN (1, 2, 3) THEN
      RAISE INVALID_NUMBER;  -- raise predefined exception
   END IF;
EXCEPTION
   WHEN INVALID_NUMBER THEN
      dbms_output.put_line('Handling invalid input by rolling back.');
      ROLLBACK;
END;
/
```

How PL/SQL Exceptions Propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception *propagates*. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. If no handler is found, PL/SQL returns an *unhandled exception* error to the host environment.

Exceptions cannot propagate across remote procedure calls done through database links. A PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see "Defining Your Own Error Messages: Procedure RAISE_APPLICATION_ERROR" on page 10-8.

Figure 10–1, Figure 10–2, and Figure 10–3 illustrate the basic propagation rules.

Figure 10–1 Propagation Rules: Example 1

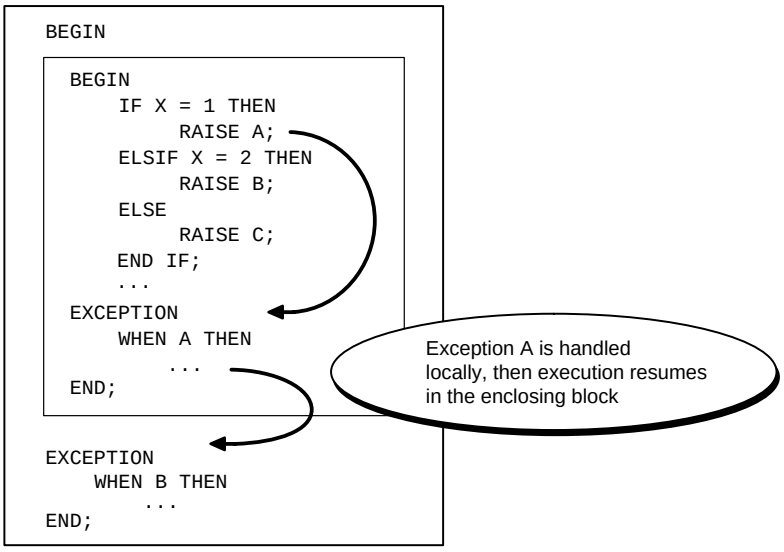


Figure 10–2 Propagation Rules: Example 2

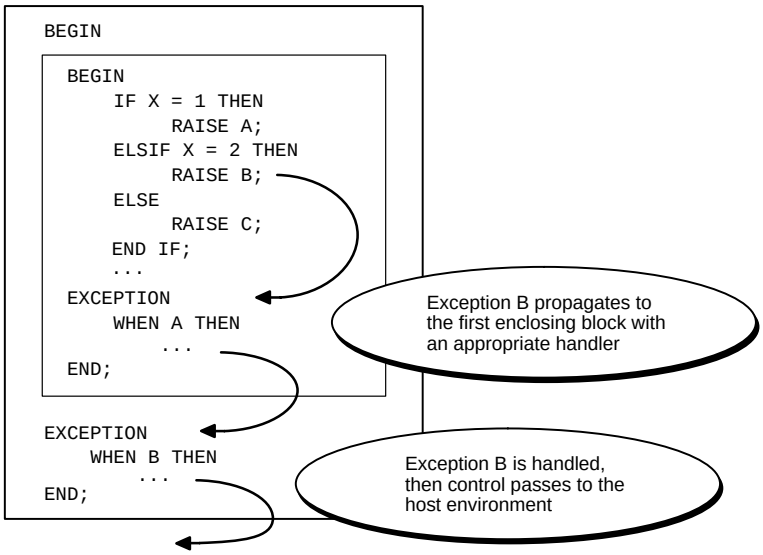
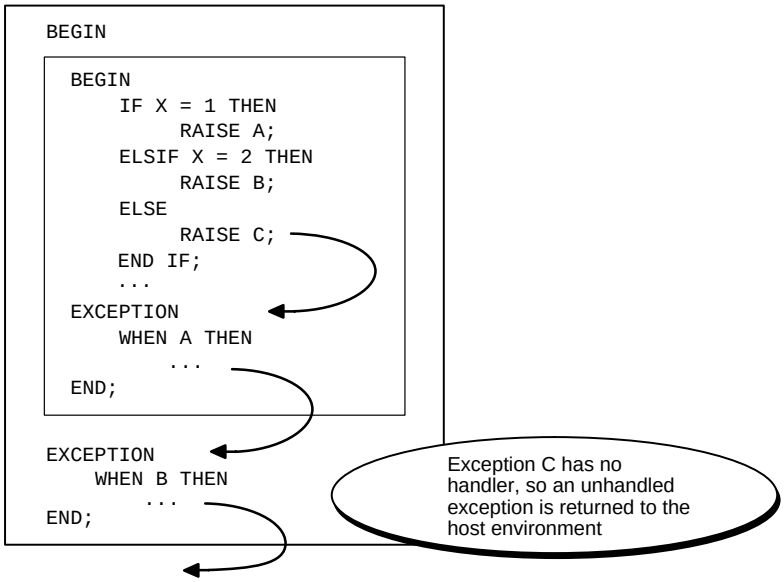


Figure 10–3 Propagation Rules: Example 3



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;
     due_date DATE := trunc(SYSDATE) - 1;
     todays_date DATE := trunc(SYSDATE);
   BEGIN
     IF due_date < todays_date THEN
        RAISE past_due;
     END IF;
   END;  ------------- sub-block ends
EXCEPTION
```

```
   WHEN OTHERS THEN
      ROLLBACK;
END;
/
```

Because the block that declares the exception `past_due` has no handler for it, the exception propagates to the enclosing block. But the enclosing block cannot reference the name `PAST_DUE`, because the scope where it was declared no longer exists. Once the exception name is lost, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the calling application gets this error:

```
ORA-06510: PL/SQL: unhandled user-defined exception
```

Reraising a PL/SQL Exception

Sometimes, you want to *reraise* an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, use a `RAISE` statement without an exception name, which is allowed only in an exception handler:

```
DECLARE
   salary_too_high  EXCEPTION;
   current_salary NUMBER := 20000;
   max_salary NUMBER := 10000;
   erroneous_salary NUMBER;
BEGIN
   BEGIN  ---------- sub-block begins
      IF current_salary > max_salary THEN
         RAISE salary_too_high;  -- raise the exception
      END IF;
   EXCEPTION
      WHEN salary_too_high THEN
         -- first step in handling the error
        dbms_output.put_line('Salary ' || erroneous_salary ||
          ' is out of range.');
        dbms_output.put_line('Maximum salary is ' || max_salary || '.');
         RAISE;  -- reraise the current exception
   END;  ------------ sub-block ends
EXCEPTION
   WHEN salary_too_high THEN
      -- handle the error more thoroughly
      erroneous_salary := current_salary;
      current_salary := max_salary;
     dbms_output.put_line('Revising salary from ' || erroneous_salary ||
       'to ' || current_salary || '.');
END;
/
```

Handling Raised PL/SQL Exceptions

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as follows:

```
EXCEPTION
   WHEN 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.

As the following example shows, use of the `OTHERS` handler guarantees that *no* exception will go unhandled:

```
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:

```
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:

```
DECLARE
   credit_limit CONSTANT NUMBER(3) := 5000;  -- raises an exception
BEGIN
   NULL;
EXCEPTION
```

```
   WHEN OTHERS THEN
      -- Cannot catch the exception. This handler is never called.
      dbms_output.put_line('Can''t handle an exception in a declaration.');
END;
/
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates *immediately* to the enclosing block.

Handling Exceptions Raised in Handlers

When an exception occurs within an exception handler, that same handler cannot catch the exception. An exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for this new exception. From there on, the exception propagates normally. For example:

```
EXCEPTION
   WHEN INVALID_NUMBER THEN
      INSERT INTO ...  -- might raise DUP_VAL_ON_INDEX
   WHEN DUP_VAL_ON_INDEX THEN ...  -- cannot catch the exception
END;
```

Branching to or from an Exception Handler

A `GOTO` statement can branch from an exception handler into an enclosing block.

A `GOTO` statement cannot branch into an exception handler, or from an exception handler into the current block.

Retrieving the Error Code and Error Message: SQLCODE and SQLERRM

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`.

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*.

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_msg VARCHAR2(100);
BEGIN
   /* Get a few Oracle error messages. */
   FOR err_num IN 1..3 LOOP
     err_msg := SUBSTR(SQLERRM(-err_num),1,100);
     dbms_output.put_line('Error number = ' || err_num);
     dbms_output.put_line('Error message = ' || err_msg);
   END LOOP;
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:

```
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
      NULL;
END;
/
```

You can still handle an exception for a statement, then continue with the next statement. Place the statement in its own sub-block with its own exception handlers. If an error occurs in the sub-block, a local handler can catch the exception. When the sub-block ends, the enclosing block continues to execute at the point where the sub-block ends. Consider the following example:

```
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);
EXCEPTION
   WHEN OTHERS THEN
      NULL;
END;
/
```

In this example, if the `SELECT INTO` statement raises a `ZERO_DIVIDE` exception, the local handler catches it and sets `pe_ratio` to zero. Execution of the handler is complete, so the sub-block terminates, and execution continues with the `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 %BULK_EXCEPTIONS Attribute" on page 11-13.

Retrying a Transaction

After an exception is raised, rather than abandon your transaction, you might want to retry it. The technique is:

1. Encase the transaction in a sub-block.
2. Place the sub-block inside a loop that repeats the transaction.
3. Before starting the transaction, mark a savepoint. If the transaction succeeds, commit, then exit from the loop. If the transaction fails, control transfers to the exception handler, where you roll back to the savepoint undoing any changes, then try to fix the problem.

In the following example, the `INSERT` statement might raise an exception because of a duplicate value in a unique column. In that case, we change the value that needs to be unique and continue with the next loop iteration. If the INSERT succeeds, we exit from the loop immediately. With this technique, you should use a `FOR` or `WHILE` loop to limit the number of attempts.

```
DECLARE
```

```
   name   VARCHAR2(20);
   ans1   VARCHAR2(3);
   ans2   VARCHAR2(3);
   ans3   VARCHAR2(3);
   suffix NUMBER := 1;
BEGIN
   FOR i IN 1..10 LOOP  -- try 10 times
      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

Using one exception handler for a sequence of statements, such as INSERT, DELETE, or UPDATE statements, can mask the statement that caused an error. If you need to know which statement failed, you can use a *locator variable*:

```
DECLARE
   stmt INTEGER;
   name VARCHAR2(100);
BEGIN
   stmt := 1;  -- designates 1st SELECT statement
   SELECT table_name INTO name FROM user_tables WHERE table_name LIKE 'ABC%';
   stmt := 2;  -- designates 2nd SELECT statement
   SELECT table_name INTO name FROM user_tables WHERE table_name LIKE 'XYZ%';
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      dbms_output.put_line('Table name not found in query ' || stmt);
END;
/
```

Overview of PL/SQL Compile-Time Warnings

To make your programs more robust and avoid problems at run time, you can turn on checking for certain warning conditions. These conditions are not serious enough to produce an error and keep you from compiling a subprogram. They might point out something in the subprogram that produces an undefined result or might create a performance problem.

To work with PL/SQL warning messages, you use the PLSQL_WARNINGS initialization parameter, the DBMS_WARNING package, and the USER/DBA/ALL_PLSQL_OBJECT_SETTINGS views.

PL/SQL Warning Categories

PL/SQL warning messages are divided into categories, so that you can suppress or display groups of similar warnings during compilation. The categories are:

Severe: Messages for conditions that might cause unexpected behavior or wrong results, such as aliasing problems with parameters.

Performance: Messages for conditions that might cause performance problems, such as passing a VARCHAR2 value to a NUMBER column in an INSERT statement.

Informational: Messages for conditions that do not have an effect on performance or correctness, but that you might want to change to make the code more maintainable, such as dead code that can never be executed.

The keyword **All** is a shorthand way to refer to all warning messages.

You can also treat particular messages as errors instead of warnings. For example, if you know that the warning message PLW-05003 represents a serious problem in your code, including 'ERROR:05003' in the PLSQL_WARNINGS setting makes that condition trigger an error message (PLS_05003) instead of a warning message. An error message causes the compilation to fail.

Controlling PL/SQL Warning Messages

To let the database issue warning messages during PL/SQL compilation, you set the initialization parameter PLSQL_WARNINGS. You can enable and disable entire categories of warnings (ALL, SEVERE, INFORMATIONAL, PERFORMANCE), enable and disable specific message numbers, and make the database treat certain warnings as compilation errors so that those conditions must be corrected.

This parameter can be set at the system level or the session level. You can also set it for a single compilation by including it as part of the ALTER PROCEDURE statement. You might turn on all warnings during development, turn off all warnings when deploying for production, or turn on some warnings when working on a particular subprogram where you are concerned with some aspect, such as unnecessary code or performance.

```
ALTER SYSTEM SET PLSQL_WARNINGS='ENABLE:ALL'; -- For debugging during development.
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE'; -- To focus on one aspect.
ALTER PROCEDURE hello COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'; -- Recompile
with extra checking.
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL'; -- To turn off all warnings.
-- We want to hear about 'severe' warnings, don't want to hear about 'performance'
-- warnings, and want PLW-06002 warnings to produce errors that halt compilation.
ALTER SESSION SET
PLSQL_WARNINGS='ENABLE:SEVERE','DISABLE:PERFORMANCE','ERROR:06002';
```

Warning messages can be issued during compilation of PL/SQL subprograms; anonymous blocks do not produce any warnings.

The settings for the PLSQL_WARNINGS parameter are stored along with each compiled subprogram. If you recompile the subprogram with a CREATE OR REPLACE statement, the current settings for that session are used. If you recompile the subprogram with an ALTER ... COMPILE statement, the current session setting might be used, or the original setting that was stored with the subprogram, depending on whether you include the REUSE SETTINGS clause in the statement.

To see any warnings generated during compilation, you use the SQL*Plus SHOW ERRORS command or query the USER_ERRORS data dictionary view. PL/SQL warning messages all use the prefix PLW.

Using the DBMS_WARNING Package

If you are writing a development environment that compiles PL/SQL subprograms, you can control PL/SQL warning messages by calling subprograms in the DBMS_WARNING package. You might also use this package when compiling a complex application, made up of several nested SQL*Plus scripts, where different warning settings apply to different subprograms. You can save the current state of the PLSQL_WARNINGS parameter with one call to the package, change the parameter to compile a particular set of subprograms, then restore the original parameter value.

For example, here is a procedure with unnecessary code that could be removed. It could represent a mistake, or it could be intentionally hidden by a debug flag, so you might or might not want a warning message for it.

```
CREATE OR REPLACE PROCEDURE dead_code
AS
  x number := 10;
BEGIN
  if x = 10 then
      x := 20;
  else
    x := 100; -- dead code (never reached)
  end if;
END dead_code;
/
-- By default, the preceding procedure compiles with no errors or warnings.

-- Now enable all warning messages, just for this session.
CALL DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:ALL' ,'SESSION');

-- Check the current warning setting.
select dbms_warning.get_warning_setting_string() from dual;

-- When we recompile the procedure, we will see a warning about the dead code.
ALTER PROCEDURE dead_code COMPILE;
```

See Also: ALTER PROCEDURE, DBMS_WARNING package in the *PL/SQL Packages and Types Reference*, PLW- messages in the *Oracle Database Error Messages*

11

Tuning PL/SQL Applications for Performance

Every day, in every way, I am getting better and better. —Émile Coué

This chapter shows you how to write efficient PL/SQL code, and speed up existing code.

This chapter contains these topics:

How PL/SQL Optimizes Your Programs

In releases prior to 10*g*, the PL/SQL compiler translated your code to machine code without applying many changes for performance. Now, PL/SQL uses an optimizing compiler that can rearrange code for better performance.

You do not need to do anything to get the benefits of this new optimizer. It is enabled by default. In rare cases, if the overhead of the optimizer makes compilation of very large applications take too long, you might lower the optimization by setting the initialization parameter `PLSQL_OPTIMIZE_LEVEL=1` instead of its default value 2. In even rarer cases, you might see a change in exception behavior, either an exception that is not raised at all, or one that is raised earlier than expected. Setting `PL_SQL_OPTIMIZE_LEVEL=0` prevents the code from being rearranged at all.

When to Tune PL/SQL Code

The information in this chapter is especially valuable if you are responsible for:

- Programs that do a lot of mathematical calculations. You will want to investigate the datatypes `PLS_INTEGER`, `BINARY_FLOAT`, and `BINARY_DOUBLE`.
- Functions that are called from PL/SQL queries, where the functions might be executed millions of times. You will want to look at all performance features to make the function as efficient as possible, and perhaps a function-based index to precompute the results for each row and save on query time.
- Programs that spend a lot of time processing `INSERT`, `UPDATE`, or `DELETE` statements, or looping through query results. You will want to investigate the `FORALL` statement for issuing DML, and the `BULK COLLECT INTO` and `RETURNING BULK COLLECT INTO` clauses for queries.
- Older code that does not take advantage of recent PL/SQL language features. (With the many performance improvements in Oracle Database 10*g*, any code from earlier releases is a candidate for tuning.)
- Any program that spends a lot of time doing PL/SQL processing, as opposed to issuing DDL statements like `CREATE TABLE` that are just passed directly to SQL. You will want to investigate native compilation. Because many built-in database features use PL/SQL, you can apply this tuning feature to an entire database to improve performance in many areas, not just your own code.

Before starting any tuning effort, benchmark the current system and measure how long particular subprograms take. PL/SQL in Oracle Database 10*g* includes many automatic optimizations, so you might see performance improvements without doing any tuning.

Guidelines for Avoiding PL/SQL Performance Problems

When a PL/SQL-based application performs poorly, it is often due to badly written SQL statements, poor programming practices, inattention to PL/SQL basics, or misuse of shared memory.

Avoiding CPU Overhead in PL/SQL Code

Make SQL Statements as Efficient as Possible

PL/SQL programs look relatively simple because most of the work is done by SQL statements. Slow SQL statements are the main reason for slow execution.

If SQL statements are slowing down your program:

- Make sure you have appropriate indexes. There are different kinds of indexes for different situations. Your index strategy might be different depending on the sizes of various tables in a query, the distribution of data in each query, and the columns used in the `WHERE` clauses.
- Make sure you have up-to-date statistics on all the tables, using the subprograms in the `DBMS_STATS` package.
- Analyze the execution plans and performance of the SQL statements, using:
  - `EXPLAIN PLAN` statement
  - SQL Trace facility with `TKPROF` utility
  - Oracle Trace facility
- Rewrite the SQL statements if necessary. For example, query hints can avoid problems such as unnecessary full-table scans.

For more information about these methods, see *Oracle Database Performance Tuning Guide*.

Some PL/SQL features also help improve the performance of SQL statements:

- If you are running SQL statements inside a PL/SQL loop, look at the FORALL statement as a way to replace loops of INSERT, UPDATE, and DELETE statements.
- If you are looping through the result set of a query, look at the BULK COLLECT clause of the SELECT INTO statement as a way to bring the entire result set into memory in a single operation.

Make Function Calls as Efficient as Possible

Badly written subprograms (for example, a slow sort or search function) can harm performance. Avoid unnecessary calls to subprograms, and optimize their code:

- If a function is called within a SQL query, you can cache the function value for each row by creating a function-based index on the table in the query. The CREATE INDEX statement might take a while, but queries can be much faster.
- If a column is passed to a function within an SQL query, the query cannot use regular indexes on that column, and the function might be called for every row in a (potentially very large) table. Consider nesting the query so that the inner query filters the results to a small number of rows, and the outer query calls the function only a few times:

```
BEGIN
-- Inefficient, calls my_function for every row.
   FOR item IN (SELECT DISTINCT(SQRT(department_id)) col_alias FROM employees)
   LOOP
      dbms_output.put_line(item.col_alias);
   END LOOP;

-- Efficient, only calls function once for each distinct value.
   FOR item IN
   ( SELECT SQRT(department_id) col_alias FROM
     ( SELECT DISTINCT department_id FROM employees)
   )
   LOOP
      dbms_output.put_line(item.col_alias);
   END LOOP;
END;
/
```

If you use OUT or IN OUT parameters, PL/SQL adds some performance overhead to ensure correct behavior in case of exceptions (assigning a value to the OUT parameter, then exiting the subprogram because of an unhandled exception, so that the OUT parameter keeps its original value).

If your program does not depend on OUT parameters keeping their values in such situations, you can add the NOCOPY keyword to the parameter declarations, so the parameters are declared OUT NOCOPY or IN OUT NOCOPY.

This technique can give significant speedup if you are passing back large amounts of data in OUT parameters, such as collections, big VARCHAR2 values, or LOBs.

This technique also applies to member subprograms of object types. If these subprograms modify attributes of the object type, all the attributes are copied when the subprogram ends. To avoid this overhead, you can explicitly declare the first parameter of the member subprogram as SELF IN OUT NOCOPY, instead of relying on PL/SQL's implicit declaration SELF IN OUT.

Make Loops as Efficient as Possible

Because PL/SQL applications are often built around loops, it is important to optimize the loop itself and the code inside the loop:

- Move initializations or computations outside the loop if possible.
- To issue a series of DML statements, replace loop constructs with `FORALL` statements.
- To loop through a result set and store the values, use the `BULK COLLECT` clause on the query to bring the query results into memory in one operation.
- If you have to loop through a result set more than once, or issue other queries as you loop through a result set, you can probably enhance the original query to give you exactly the results you want. Some query operators to explore include `UNION`, `INTERSECT`, `MINUS`, and `CONNECT BY`.
- You can also nest one query inside another (known as a subselect) to do the filtering and sorting in multiple stages. For example, instead of calling a PL/SQL function in the inner `WHERE` clause (which might call the function once for each row of the table), you can filter the result set to a small set of rows in the inner query, and call the function in the outer query.

Don't Duplicate Built-in String Functions

PL/SQL provides many highly optimized string functions such as `REPLACE`, `TRANSLATE`, `SUBSTR`, `INSTR`, `RPAD`, and `LTRIM`. The built-in functions use low-level code that is more efficient than regular PL/SQL.

If you use PL/SQL string functions to search for regular expressions, consider using the built-in regular expression functions, such as `REGEXP_SUBSTR`.

Reorder Conditional Tests to Put the Least Expensive First

PL/SQL stops evaluating a logical expression as soon as the result can be determined (known as short-circuit evaluation).

When evaluating multiple conditions separated by `AND` or `OR`, put the least expensive ones first. For example, check the values of PL/SQL variables before testing function return values, because PL/SQL might be able to skip calling the functions.

Minimize Datatype Conversions

At run time, PL/SQL converts between different datatypes automatically. For example, assigning a `PLS_INTEGER` variable to a `NUMBER` variable results in a conversion because their internal representations are different.

Avoiding implicit conversions can improve performance. Use literals of the appropriate types: character literals in character expressions, decimal numbers in number expressions, and so on.

In the example below, the integer literal `15` must be converted to an Oracle `NUMBER` before the addition. The floating-point literal `15.0` is represented as a `NUMBER`, avoiding the need for a conversion.

```
DECLARE
   n NUMBER;
   c CHAR(5);
BEGIN
   n := n + 15;      -- converted implicitly; slow
   n := n + 15.0;    -- not converted; fast
   c := 25;          -- converted implicitly; slow
```

```
   c := TO_CHAR(25); -- converted explicitly; still slow
   c := '25';        -- not converted; fast
END;
/
```

Minimizing conversions might mean changing the types of your variables, or even working backward and designing your tables with different datatypes. Or, you might convert data once (such as from an `INTEGER` column to a `PLS_INTEGER` variable) and use the PL/SQL type consistently after that.

Use PLS_INTEGER or BINARY_INTEGER for Integer Arithmetic

When you need to declare a local integer variable, use the datatype `PLS_INTEGER`, which is the most efficient integer type. `PLS_INTEGER` values require less storage than `INTEGER` or `NUMBER` values, and `PLS_INTEGER` operations use machine arithmetic.

The `BINARY_INTEGER` datatype is just as efficient as `PLS_INTEGER` for any new code, but if you are running the same code on Oracle9*i* or Oracle8*i* databases, `PLS_INTEGER` is faster.

The datatype `NUMBER` and its subtypes are represented in a special internal format, designed for portability and arbitrary scale and precision, not performance. Even the subtype `INTEGER` is treated as a floating-point number with nothing after the decimal point. Operations on `NUMBER` or `INTEGER` variables require calls to library routines.

Avoid constrained subtypes such as `INTEGER`, `NATURAL`, `NATURALN`, `POSITIVE`, `POSITIVEN`, and `SIGNTYPE` in performance-critical code. Variables of these types require extra checking at run time, each time they are used in a calculation.

Use BINARY_FLOAT and BINARY_DOUBLE for Floating-Point Arithmetic

The datatype `NUMBER` and its subtypes are represented in a special internal format, designed for portability and arbitrary scale and precision, not performance. Operations on `NUMBER` or `INTEGER` variables require calls to library routines.

The `BINARY_FLOAT` and `BINARY_DOUBLE` types can use native machine arithmetic instructions, and are more efficient for number-crunching applications such as scientific processing. They also require less space in the database.

These types do not always represent fractional values precisely, and handle rounding differently than the `NUMBER` types. These types are less suitable for financial code where accuracy is critical.

Avoiding Memory Overhead in PL/SQL Code

Be Generous When Declaring Sizes for VARCHAR2 Variables

You might need to allocate large `VARCHAR2` variables when you are not sure how big an expression result will be. You can actually conserve memory by declaring `VARCHAR2` variables with large sizes, such as 32000, rather than estimating just a little on the high side, such as by specifying a size such as 256 or 1000. PL/SQL has an optimization that makes it easy to avoid overflow problems and still conserve memory. Specify a size of 2000 or more characters for the `VARCHAR2` variable; PL/SQL waits until you assign the variable, then only allocates as much storage as needed.

Group Related Subprograms into Packages

When you call a packaged subprogram for the first time, the whole package is loaded into the shared memory pool. Subsequent calls to related subprograms in the package

require no disk I/O, and your code executes faster. If the package is aged out of memory, it must be reloaded if you reference it again.

You can improve performance by sizing the shared memory pool correctly. Make sure it is large enough to hold all frequently used packages but not so large that memory is wasted.

Pin Packages in the Shared Memory Pool

You can "pin" frequently accessed packages in the shared memory pool, using the supplied package `DBMS_SHARED_POOL`. When a package is pinned, it is not aged out by the least recently used (LRU) algorithm that Oracle normally uses. The package remains in memory no matter how full the pool gets or how frequently you access the package.

For more information on the `DBMS_SHARED_POOL` package, see *PL/SQL Packages and Types Reference*.

Improve Your Code to Avoid Compiler Warnings

The PL/SQL compiler issues warnings about things that do not make a program incorrect, but might lead to poor performance. If you receive such a warning, and the performance of this code is important, follow the suggestions in the warning and change the code to be more efficient.

Profiling and Tracing PL/SQL Programs

As you develop larger and larger PL/SQL applications, it becomes more difficult to isolate performance problems. 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.

Using 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.

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 information about the `DBMS_PROFILER` subprograms, see *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.

Using 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.

For information about the `DBMS_TRACE` subprograms, see *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.

Reducing Loop Overhead for DML Statements and Queries (FORALL, BULK COLLECT)

PL/SQL sends SQL statements such as DML and queries to the SQL engine for execution, and SQL returns the result data to PL/SQL. You can minimize the performance overhead of this communication between PL/SQL and SQL by using the PL/SQL language features known collectively as bulk SQL. The `FORALL` statement sends `INSERT`, `UPDATE`, or `DELETE` statements in batches, rather than one at a time. The `BULK COLLECT` clause brings back batches of results from SQL. If the DML statement affects four or more database rows, the use of bulk SQL can improve performance considerably.

The assigning of values to PL/SQL variables in SQL statements is called **binding**. PL/SQL binding operations fall into three categories:

- **in-bind** When a PL/SQL variable or host variable is stored in the database by an `INSERT` or `UPDATE` statement.
- **out-bind** When a database value is assigned to a PL/SQL variable or a host variable by the `RETURNING` clause of an `INSERT`, `UPDATE`, or `DELETE` statement.
- **define** When a database value is assigned to a PL/SQL variable or a host variable by a `SELECT` or `FETCH` statement.

Bulk SQL uses PL/SQL collections, such as varrays or nested tables, to pass large amounts of data back and forth in a single operation. This process is known as **bulk binding**. If the collection has 20 elements, bulk binding lets you perform the equivalent of 20 `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements using a single

operation. Queries can pass back any number of results, without requiring a FETCH statement for each row.

To speed up INSERT, UPDATE, and DELETE statements, enclose the SQL statement within a PL/SQL FORALL statement instead of a loop construct.

To speed up SELECT statements, include the BULK COLLECT INTO clause in the SELECT statement instead of using INTO.

For full details of the syntax and restrictions for these statements, see "FORALL Statement" on page 13-64 and "SELECT INTO Statement" on page 13-123.

Using the FORALL Statement

The keyword FORALL lets you run multiple DML statements very efficiently. It can only repeat a single DML statement, unlike a general-purpose FOR loop.

For full syntax and restrictions, see "FORALL Statement" on page 13-64.

The SQL statement can reference more than one collection, but FORALL only improves performance where the index value is used as a subscript.

Usually, the bounds specify a range of consecutive index numbers. If the index numbers are not consecutive, such as after you delete collection elements, you can use the INDICES OF or VALUES OF clause to iterate over just those index values that really exist.

The INDICES OF clause iterates over all of the index values in the specified collection, or only those between a lower and upper bound.

The VALUES OF clause refers to a collection that is indexed by BINARY_INTEGER or PLS_INTEGER and whose elements are of type BINARY_INTEGER or PLS_INTEGER. The FORALL statement iterates over the index values specified by the elements of this collection.

Example 11–1 Issuing DELETE Statements in a Loop

This FORALL statement sends all three DELETE statements to the SQL engine at once:

```
CREATE TABLE employees2 AS SELECT * FROM employees;
DECLARE
   TYPE NumList IS VARRAY(20) OF NUMBER;
   depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
   FORALL i IN depts.FIRST..depts.LAST
      DELETE FROM employees2 WHERE department_id = depts(i);
   COMMIT;
END;
/
DROP TABLE employees2;
```

Example 11–2 Issuing INSERT Statements in a Loop

The following example loads some data into PL/SQL collections. Then it inserts the collection elements into a database table twice: first using a FOR loop, then using a FORALL statement. The FORALL version is much faster.

```
CREATE TABLE parts1 (pnum INTEGER, pname VARCHAR2(15));
CREATE TABLE parts2 (pnum INTEGER, pname VARCHAR2(15));
DECLARE
  TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
```

```
  TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
  pnums  NumTab;
  pnames NameTab;
  iterations CONSTANT PLS_INTEGER := 500;
  t1 INTEGER;  t2 INTEGER;  t3 INTEGER;
BEGIN
  FOR j IN 1..iterations LOOP  -- load index-by tables
     pnums(j) := j;
     pnames(j) := 'Part No. ' || TO_CHAR(j);
  END LOOP;
  t1 := dbms_utility.get_time;
  FOR i IN 1..iterations LOOP  -- use FOR loop
     INSERT INTO parts1 VALUES (pnums(i), pnames(i));
  END LOOP;
  t2 := dbms_utility.get_time;
  FORALL i IN 1..iterations  -- use FORALL statement
     INSERT INTO parts2 VALUES (pnums(i), pnames(i));
  t3 := dbms_utility.get_time;
  dbms_output.put_line('Execution Time (secs)');
  dbms_output.put_line('--------------------');
  dbms_output.put_line('FOR loop: ' || TO_CHAR((t2 - t1)/100));
  dbms_output.put_line('FORALL:   ' || TO_CHAR((t3 - t2)/100));
  COMMIT;
END;
/
DROP TABLE parts1;
DROP TABLE parts2;
```

Executing this block should show that the loop using `FORALL` is much faster.

Example 11–3 Using FORALL with Part of a Collection

The bounds of the `FORALL` loop can apply to part of a collection, not necessarily all the elements:

```
CREATE TABLE employees2 AS SELECT * FROM employees;
DECLARE
   TYPE NumList IS VARRAY(10) OF NUMBER;
   depts NumList := NumList(5,10,20,30,50,55,57,60,70,75);
BEGIN
   FORALL j IN 4..7  -- use only part of varray
      DELETE FROM employees2 WHERE department_id = depts(j);
   COMMIT;
END;
/
DROP TABLE employees2;
```

Example 11–4 Using FORALL with Non-Consecutive Index Values

You might need to delete some elements from a collection before using the collection in a `FORALL` statement. The `INDICES OF` clause processes sparse collections by iterating through only the remaining elements.

You might also want to leave the original collection alone, but process only some elements, process the elements in a different order, or process some elements more than once. Instead of copying the entire elements into new collections, which might use up substantial amounts of memory, the `VALUES OF` clause lets you set up simple collections whose elements serve as "pointers" to elements in the original collection.

The following example creates a collection holding some arbitrary data, a set of table names. Deleting some of the elements makes it a sparse collection that would not work in a default FORALL statement. The program uses a FORALL statement with the INDICES OF clause to insert the data into a table. It then sets up two more collections, pointing to certain elements from the original collection. The program stores each set of names in a different database table using FORALL statements with the VALUES OF clause.

```
-- Create empty tables to hold order details
CREATE TABLE valid_orders (cust_name VARCHAR2(32), amount NUMBER(10,2));
CREATE TABLE big_orders AS SELECT * FROM valid_orders WHERE 1 = 0;
CREATE TABLE rejected_orders AS SELECT * FROM valid_orders WHERE 1 = 0;

DECLARE
-- Make collections to hold a set of customer names and order amounts.

   SUBTYPE cust_name IS valid_orders.cust_name%TYPE;
   TYPE cust_typ IS TABLe OF cust_name;
   cust_tab cust_typ;

   SUBTYPE order_amount IS valid_orders.amount%TYPE;
   TYPE amount_typ IS TABLE OF NUMBER;
   amount_tab amount_typ;

-- Make other collections to point into the CUST_TAB collection.
   TYPE index_pointer_t IS TABLE OF PLS_INTEGER;
   big_order_tab index_pointer_t := index_pointer_t();
   rejected_order_tab index_pointer_t := index_pointer_t();

   PROCEDURE setup_data IS BEGIN
      -- Set up sample order data, including some invalid orders and some 'big'
orders.
      cust_tab := cust_typ('Company 1','Company 2','Company 3','Company 4',
'Company 5');
      amount_tab := amount_typ(5000.01, 0, 150.25, 4000.00, NULL);
   END;

BEGIN
   setup_data();

   dbms_output.put_line('--- Original order data ---');
   FOR i IN 1..cust_tab.LAST LOOP
      dbms_output.put_line('Customer #' || i || ', ' || cust_tab(i) || ': $' ||
amount_tab(i));
   END LOOP;

-- Delete invalid orders (where amount is null or 0).
   FOR i IN 1..cust_tab.LAST LOOP
      IF amount_tab(i) is null or amount_tab(i) = 0 THEN
         cust_tab.delete(i);
         amount_tab.delete(i);
      END IF;
   END LOOP;

   dbms_output.put_line('--- Data with invalid orders deleted ---');
   FOR i IN 1..cust_tab.LAST LOOP
      IF cust_tab.EXISTS(i) THEN
         dbms_output.put_line('Customer #' || i || ', ' || cust_tab(i) || ': $' ||
amount_tab(i));
      END IF;
```

```
   END LOOP;

-- Since the subscripts of our collections are not consecutive, we use
-- FORALL...INDICES OF to iterate through the actual subscripts, rather than
1..COUNT.
   FORALL i IN INDICES OF cust_tab
      INSERT INTO valid_orders(cust_name, amount) VALUES(cust_tab(i),
amount_tab(i));

-- Now let's process the order data differently. We'll extract 2 subsets
-- and store each subset in a different table.

   setup_data(); -- Initialize the CUST_TAB and AMOUNT_TAB collections again.

   FOR i IN cust_tab.FIRST .. cust_tab.LAST LOOP
      IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
         rejected_order_tab.EXTEND; -- Add a new element to this collection.
         rejected_order_tab(rejected_order_tab.LAST) := i; -- And record the
subscript from the original collection.
      END IF;
      IF amount_tab(i) > 2000 THEN
         big_order_tab.EXTEND; -- Add a new element to this collection.
         big_order_tab(big_order_tab.LAST) := i; -- And record the subscript from
the original collection.
      END IF;
   END LOOP;

-- Now it's easy to run one DML statement on one subset of elements, and another
DML statement on a different subset.

   FORALL i IN VALUES OF rejected_order_tab
      INSERT INTO rejected_orders VALUES (cust_tab(i), amount_tab(i));

   FORALL i IN VALUES OF big_order_tab
      INSERT INTO big_orders VALUES (cust_tab(i), amount_tab(i));

   COMMIT;
END;
/
-- Verify that the correct order details were stored.
SELECT cust_name "Customer", amount "Valid order amount"  FROM valid_orders;
SELECT cust_name "Customer", amount "Big order amount"  FROM big_orders;
SELECT cust_name "Customer", amount "Rejected order amount"  FROM rejected_orders;

DROP TABLE valid_orders;
DROP TABLE big_orders;
DROP TABLE rejected_orders;
```

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. Then, you change the job titles so that they are longer. The second UPDATE fails because the new value is too long for the column. Because we handle the exception, the first UPDATE is not rolled back and we can commit that change.

```
CREATE TABLE emp2 (deptno NUMBER(2), job VARCHAR2(18));
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(10, 20, 30);
BEGIN
   INSERT INTO emp2 VALUES(10, 'Clerk');
   INSERT INTO emp2 VALUES(20, 'Bookkeeper');  -- Lengthening this job title
causes an exception.
   INSERT INTO emp2 VALUES(30, 'Analyst');
   COMMIT;

   FORALL j IN depts.FIRST..depts.LAST -- Run 3 UPDATE statements.
      UPDATE emp2 SET job = job || ' (Senior)' WHERE deptno = depts(j);
         -- raises a "value too large" exception
EXCEPTION
   WHEN OTHERS THEN
      dbms_output.put_line('Problem in the FORALL statement.');
      COMMIT; -- Commit results of successful updates.
END;
/
DROP TABLE emp2;
```

Counting Rows Affected by FORALL with the %BULK_ROWCOUNT Attribute

The cursor attributes SQL%FOUND, SQL%ISOPEN, SQL%NOTFOUND, and SQL%ROWCOUNT, return useful information about the most recently executed DML statement.

The SQL cursor has one composite attribute, %BULK_ROWCOUNT, for use with the FORALL statement. This attribute works like an associative array: SQL%BULK_ROWCOUNT(*i*) stores the number of rows processed by the *i*th execution of an INSERT, UPDATE or DELETE statement. For example:

```
CREATE TABLE emp2 AS SELECT * FROM employees;
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(30, 50, 60);
BEGIN
   FORALL j IN depts.FIRST..depts.LAST
      DELETE FROM emp2 WHERE department_id = depts(j);
-- How many rows were affected by each DELETE statement?
   FOR i IN depts.FIRST..depts.LAST
   LOOP
      dbms_output.put_line('Iteration #' || i || ' deleted ' ||
         SQL%BULK_ROWCOUNT(i) || ' rows.');
   END LOOP;
END;
/
DROP TABLE emp2;
```

The FORALL statement and %BULK_ROWCOUNT attribute use the same subscripts. For example, if FORALL uses the range 5..10, so does %BULK_ROWCOUNT. If the FORALL tatement uses the INDICES OF clause to process a sparse collection, %BULK_ROWCOUNT has corresponding sparse subscripts. If the FORALL statement uses the VALUES OF clause to process a subset of elements, %BULK_ROWCOUNT has subscripts corresponding to the values of the elements in the index collection. If the index collection contains duplicate elements, so that some DML statements are issued multiple times using the same subscript, then the corresponding elements of %BULK_ROWCOUNT represent the sum of all rows affected by the DML statement using

that subscript. (For examples showing how to interpret %BULK_ROWCOUNT when using the INDICES OF and VALUES OF clauses, see the PL/SQL sample programs at http://otn.oracle.com/tech/pl_sql/.)

%BULK_ROWCOUNT is usually equal to 1 for inserts, because a typical insert operation affects only a single row. For the INSERT ... SELECT construct, %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:

```
CREATE TABLE emp_by_dept AS SELECT employee_id, department_id
   FROM employees WHERE 1 = 0;
DECLARE
  TYPE dept_tab IS TABLE OF departments.department_id%TYPE;
  deptnums dept_tab;
BEGIN
  SELECT department_id BULK COLLECT INTO deptnums FROM departments;

  FORALL i IN 1..deptnums.COUNT
     INSERT INTO emp_by_dept
        SELECT employee_id, department_id FROM employees
           WHERE department_id = deptnums(i);

  FOR i IN 1..deptnums.COUNT LOOP
-- Count how many rows were inserted for each department; that is,
-- how many employees are in each department.
     dbms_output.put_line('Dept '||deptnums(i)||': inserted '||
                          SQL%BULK_ROWCOUNT(i)||' records');
  END LOOP;

  dbms_output.put_line('Total records inserted =' || SQL%ROWCOUNT);
END;
/
DROP TABLE emp_by_dept;
```

You can also use the scalar attributes %FOUND, %NOTFOUND, and %ROWCOUNT after running a FORALL statement. For example, %ROWCOUNT returns the total number of rows processed by all executions of the SQL statement.

%FOUND and %NOTFOUND refer only to the last execution of the SQL statement. You can use %BULK_ROWCOUNT to infer their values for individual executions. For example, when %BULK_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

Handling FORALL Exceptions 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 after the bounds, before the DML statement.

All exceptions raised during the execution are saved in the cursor attribute %BULK_EXCEPTIONS, which stores a collection of records. Each record has two fields:

- %BULK_EXCEPTIONS(i).ERROR_INDEX holds the "iteration" of the FORALL statement during which the exception was raised.
- %BULK_EXCEPTIONS(i).ERROR_CODE holds the corresponding Oracle error code.

The values stored by %BULK_EXCEPTIONS always refer to the most recently executed FORALL statement. The number of exceptions is saved in %BULK_EXCEPTIONS.COUNT. Its subscripts range from 1 to COUNT.

You might need to work backward to determine which collection element was used in the iteration that caused an exception. For example, if you use the INDICES OF clause to process a sparse collection, you must step through the elements one by one to find the one corresponding to %BULK_EXCEPTIONS(i).ERROR_INDEX. If you use the VALUES OF clause to process a subset of elements, you must find the element in the index collection whose subscript matches %BULK_EXCEPTIONS(i).ERROR_INDEX, and then use that element's value as the subscript to find the erroneous element in the original collection. (For examples showing how to find the erroneous elements when using the INDICES OF and VALUES OF clauses, see the PL/SQL sample programs at http://otn.oracle.com/tech/pl_sql/.)

If you omit the keywords SAVE EXCEPTIONS, execution of the FORALL statement stops when an exception is raised. In that case, SQL%BULK_EXCEPTIONS.COUNT returns 1, and SQL%BULK_EXCEPTIONS contains just one record. If no exception is raised during execution, SQL%BULK_EXCEPTIONS.COUNT returns 0.

Example 11–5 Bulk Operation That Continues Despite Exceptions

The following example shows how you can perform a number of DML operations, without stopping if some operations encounter errors:

```
CREATE TABLE emp2 AS SELECT * FROM employees;
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
-- The zeros in this list will cause divide-by-zero errors.
   num_tab NumList := NumList(10,0,11,12,30,0,20,199,2,0,9,1);
   errors  NUMBER;
   dml_errors EXCEPTION;
   PRAGMA exception_init(dml_errors, -24381);
BEGIN
-- SAVE EXCEPTIONS means don't stop if some DELETEs fail.
   FORALL i IN num_tab.FIRST..num_tab.LAST SAVE EXCEPTIONS
      DELETE FROM emp2 WHERE salary > 500000/num_tab(i);
-- If any errors occurred during the FORALL SAVE EXCEPTIONS,
-- a single exception is raised when the statement completes.
EXCEPTION
  WHEN dml_errors THEN -- Now we figure out what failed and why.
   errors := SQL%BULK_EXCEPTIONS.COUNT;
   dbms_output.put_line('Number of DELETE statements that failed: ' || errors);
   FOR i IN 1..errors LOOP
      dbms_output.put_line('Error #' || i || ' occurred during '||
         'iteration #' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
      dbms_output.put_line('Error message is ' ||
         SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
   END LOOP;
END;
/
DROP TABLE emp2;
```

In this example, PL/SQL raised the predefined exception ZERO_DIVIDE when i equaled 2, 6, 10. After the FORALL statement, 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 negated the value of SQL%BULK_EXCEPTIONS(i).ERROR_CODE and passed the result to the

error-reporting function SQLERRM, which expects a negative number. 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
```

Retrieving Query Results into Collections with the BULK COLLECT Clause

Using the keywords BULK COLLECT with a query is a very efficient way to retrieve the result set. Instead of looping through each row, you store the results in one or more collections, in a single operation. You can use these keywords in the SELECT INTO and FETCH INTO statements, and the RETURNING INTO clause.

With the BULK COLLECT clause, all the variables in the INTO list must be collections. The table columns can hold scalar or composite values, including object types. The following example loads two entire database columns into nested tables:

```
DECLARE
   TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
   TYPE NameTab IS TABLE OF employees.last_name%TYPE;
   enums NumTab;   -- No need to initialize the collections.
   names NameTab;  -- Values will be filled in by the SELECT INTO.
   PROCEDURE print_results IS
   BEGIN
      dbms_output.put_line('Results:');
      FOR i IN enums.FIRST .. enums.LAST
      LOOP
         dbms_output.put_line('  Employee #' || enums(i) || ': ' ||
            names(i));
      END LOOP;
   END;
BEGIN
   SELECT employee_id, last_name -- Retrieve data for 10 arbitrary employees.
      BULK COLLECT INTO enums, names
      FROM employees WHERE ROWNUM < 11;
-- The data has all been brought into memory by BULK COLLECT.
-- No need to FETCH each row from the result set.
   print_results;

   SELECT employee_id, last_name -- Retrieve approximately 20% of all rows
      BULK COLLECT INTO enums, names
      FROM employees SAMPLE (20);
   print_results;
END;
/
```

The collections are initialized automatically. Nested tables and associative arrays are extended to hold as many elements as needed. If you use varrays, all the return values must fit in the varray's declared size. Elements are inserted starting at index 1, overwriting any existing elements.

Since the processing of the BULK COLLECT INTO clause is similar to a FETCH loop, it does not raise a NO_DATA_FOUND exception if no rows match the query. You must check whether the resulting nested table or varray is null, or if the resulting associative array has no elements.

To prevent the resulting collections from expanding without limit, you can use the pseudocolumn ROWNUM to limit the number of rows processed. Or, you can use the SAMPLE clause to retrieve a random sample of rows.

```
DECLARE
   TYPE SalList IS TABLE OF emp.sal%TYPE;
   sals SalList;
BEGIN
-- Limit the number of rows to 100.
   SELECT sal BULK COLLECT INTO sals FROM emp
      WHERE ROWNUM <= 100;
-- Retrieve 10% (approximately) of the rows in the table.
   SELECT sal BULK COLLECT INTO sals FROM emp SAMPLE 10;

END;
/
```

You can process very large result sets by fetching a specified number of rows at a time from a cursor, as shown in the following sections.

Examples of Bulk-Fetching from a Cursor

Example 11–6 Bulk-Fetching from a Cursor Into One or More Collections

You can fetch from a cursor into one or more collections:

```
DECLARE
   TYPE NameList IS TABLE OF employees.last_name%TYPE;
   TYPE SalList IS TABLE OF employees.salary%TYPE;
   CURSOR c1 IS SELECT last_name, salary FROM employees WHERE salary > 10000;
   names NameList;
   sals  SalList;
   TYPE RecList IS TABLE OF c1%ROWTYPE;
   recs RecList;

   PROCEDURE print_results IS
   BEGIN
      dbms_output.put_line('Results:');
      IF names IS NULL OR names.COUNT = 0 THEN
         RETURN; -- Don't print anything if collections are empty.
      END IF;
      FOR i IN names.FIRST .. names.LAST
      LOOP
         dbms_output.put_line('  Employee ' || names(i) || ': $' ||
            sals(i));
      END LOOP;
   END;
BEGIN
   dbms_output.put_line('--- Processing all results at once ---');
   OPEN c1;
   FETCH c1 BULK COLLECT INTO names, sals;
   CLOSE c1;
   print_results;

   dbms_output.put_line('--- Processing 7 rows at a time ---');
   OPEN c1;
   LOOP
      FETCH c1 BULK COLLECT INTO names, sals LIMIT 7;
      EXIT WHEN c1%NOTFOUND;
      print_results;
```

```
   END LOOP;
-- Loop exits when fewer than 7 rows are fetched. Have to
-- process the last few. Need extra checking inside PRINT_RESULTS
-- in case it is called when the collection is empty.
   print_results;
   CLOSE c1;

   dbms_output.put_line('--- Fetching records rather than columns ---');
   OPEN c1;
   FETCH c1 BULK COLLECT INTO recs;
   FOR i IN recs.FIRST .. recs.LAST
   LOOP
-- Now all the columns from the result set come from a single record.
      dbms_output.put_line('  Employee ' || recs(i).last_name || ': $'
         || recs(i).salary);
   END LOOP;
END;
/
```

Example 11–7 Bulk-Fetching from a Cursor Into a Collection of Records

You can fetch from a cursor into a collection of records:

```
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;
END;
/
```

Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause

The optional `LIMIT` clause, allowed only in bulk `FETCH` statements, limits the number of rows fetched from the database.

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.

```
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:

```
CREATE TABLE emp2 AS SELECT * FROM employees;
DECLARE
   TYPE NumList IS TABLE OF employees.employee_id%TYPE;
   enums NumList;
   TYPE NameList IS TABLE OF employees.last_name%TYPE;
   names NameList;
BEGIN
   DELETE FROM emp2 WHERE department_id = 30
      RETURNING employee_id, last_name BULK COLLECT INTO enums, names;
   dbms_output.put_line('Deleted ' || SQL%ROWCOUNT || ' rows:');
   FOR i IN enums.FIRST .. enums.LAST
   LOOP
      dbms_output.put_line('Employee #' || enums(i) || ': ' || names(i));
   END LOOP;
END;
/
DROP TABLE emp2;
```

Using FORALL and BULK COLLECT Together

You can combine the BULK COLLECT clause with a FORALL statement. The output collections are built up as the FORALL statement iterates.

In the following example, the EMPNO value of each deleted row is stored in the collection ENUMS. The collection DEPTS has 3 elements, so the FORALL statement iterates 3 times. If each DELETE issued by the FORALL statement deletes 5 rows, then the collection ENUMS, which stores values from the deleted rows, has 15 elements when the statement completes:

```
CREATE TABLE emp2 AS SELECT * FROM employees;
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(10,20,30);
   TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
   TYPE dept_t IS TABLE OF employees.department_id%TYPE;
   e_ids enum_t;
   d_ids dept_t;
BEGIN
   FORALL j IN depts.FIRST..depts.LAST
      DELETE FROM emp2 WHERE department_id = depts(j)
         RETURNING employee_id, department_id BULK COLLECT INTO e_ids, d_ids;
   dbms_output.put_line('Deleted ' || SQL%ROWCOUNT || ' rows:');
   FOR i IN e_ids.FIRST .. e_ids.LAST
   LOOP
      dbms_output.put_line('Employee #' || e_ids(i) || ' from dept #' ||
d_ids(i));
   END LOOP;
END;
/
DROP TABLE emp2;
```

The column values returned by each execution are added to the values returned previously. If you use a FOR loop instead of the FORALL statement, the set of returned values is overwritten by each DELETE statement.

You cannot use the SELECT ... BULK COLLECT statement in a FORALL statement.

Using Host Arrays with Bulk Binds

Client-side programs can use anonymous PL/SQL blocks to bulk-bind input and output host arrays. This is the most efficient way to pass collections to and from the database server.

Host arrays are declared in a host environment such as an OCI or a Pro*C program and must be prefixed with a colon to distinguish them from PL/SQL collections. In the 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.

```
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 employees WHERE department_id = :depts(i);
   COMMIT;
END;
/
```

Writing Computation-Intensive Programs in PL/SQL

The BINARY_FLOAT and BINARY_DOUBLE datatypes make it practical to write PL/SQL programs to do number-crunching, for scientific applications involving floating-point calculations. These datatypes behave much like the native floating-point types on many hardware systems, with semantics derived from the IEEE-754 floating-point standard.

The way these datatypes represent decimal data make them less suitable for financial applications, where precise representation of fractional amounts is more important than pure performance.

The PLS_INTEGER and BINARY_INTEGER datatypes are PL/SQL-only datatypes that are more efficient than the SQL datatypes NUMBER or INTEGER for integer arithmetic. You can use PLS_INTEGER to write pure PL/SQL code for integer arithmetic, or convert NUMBER or INTEGER values to PLS_INTEGER for manipulation by PL/SQL.

In previous releases, PLS_INTEGER was more efficient than BINARY_INTEGER. Now, they have similar performance, but you might still prefer PLS_INTEGER if your code might be run under older database releases.

Within a package, you can write overloaded versions of procedures and functions that accept different numeric parameters. The math routines can be optimized for each kind of parameter (BINARY_FLOAT, BINARY_DOUBLE, NUMBER, PLS_INTEGER), avoiding unnecessary conversions.

The built-in math functions such as SQRT, SIN, COS, and so on already have fast overloaded versions that accept BINARY_FLOAT and BINARY_DOUBLE parameters. You can speed up math-intensive code by passing variables of these types to such functions, and by calling the TO_BINARY_FLOAT or TO_BINARY_DOUBLE functions when passing expressions to such functions.

Tuning Dynamic SQL with EXECUTE IMMEDIATE and Cursor Variables

Some programs (a general-purpose report writer for example) must build and process a variety of SQL statements, where the exact text of the statement is unknown until

run time. Such statements probably change from execution to execution. They are called *dynamic* SQL statements.

Formerly, to execute dynamic SQL statements, you had to use the supplied package DBMS_SQL. Now, within PL/SQL, you can execute any kind of dynamic SQL statement using an interface called *native dynamic SQL*. The main PL/SQL features involved are the EXECUTE IMMEDIATE statement and cursor variables (also known as REF CURSORs).

Native dynamic SQL code is more compact and much faster than calling the DBMS_SQL package. The following example declares a cursor variable, then associates it with a dynamic SELECT statement:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv   EmpCurTyp;
   my_ename VARCHAR2(15);
   my_sal   NUMBER := 1000;
   table_name VARCHAR2(30) := 'employees';
BEGIN
   OPEN emp_cv FOR 'SELECT last_name, salary FROM ' || table_name ||
      ' WHERE salary > :s' USING my_sal;
   CLOSE emp_cv;
END;
/
```

For more information, see Chapter 7.

Tuning PL/SQL Procedure Calls with the NOCOPY Compiler Hint

By default, OUT and IN OUT parameters are passed by value. The values of any IN OUT parameters are copied before the subprogram is executed. During subprogram execution, temporary variables hold the output parameter values. If the subprogram exits normally, these values are copied to the actual parameters. If the subprogram exits with an unhandled exception, the original parameters are unchanged.

When the parameters represent large data structures such as collections, records, and instances of object types, this copying slows down execution and uses up memory. In particular, this overhead applies to each call to an object method: temporary copies are made of all the attributes, so that any changes made by the method are only applied if the method exits normally.

To avoid this overhead, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. If the subprogram exits normally, the behavior is the same as normal. If the subprogram exits early with an exception, the values of OUT and IN OUT parameters (or object attributes) might still change. To use this technique, ensure that the subprogram handles all exceptions.

The following example asks the compiler to pass IN OUT parameter MY_STAFF by reference, to avoid copying the varray on entry to and exit from the subprogram:

```
DECLARE
   TYPE Staff IS VARRAY(200) OF Employee;
   PROCEDURE reorganize (my_staff IN OUT NOCOPY Staff) IS ...
BEGIN
   NULL;
END;
/
```

The following example loads 25,000 records into a local nested table, which is passed to two local procedures that do nothing. A call to the procedure that uses NOCOPY takes much less time.

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE;
   emp_tab EmpTabTyp := EmpTabTyp(NULL);  -- initialize
   t1 NUMBER;
   t2 NUMBER;
   t3 NUMBER;
   PROCEDURE get_time (t OUT NUMBER) IS
     BEGIN t := dbms_utility.get_time; END;
   PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
     BEGIN NULL; END;
   PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
     BEGIN NULL; END;
BEGIN
   SELECT * INTO emp_tab(1) FROM employees WHERE employee_id = 100;
   emp_tab.EXTEND(49999, 1);  -- copy element 1 into 2..50000
   get_time(t1);
   do_nothing1(emp_tab);  -- pass IN OUT parameter
   get_time(t2);
   do_nothing2(emp_tab);  -- pass IN OUT NOCOPY parameter
   get_time(t3);
   dbms_output.put_line('Call Duration (secs)');
   dbms_output.put_line('--------------------');
   dbms_output.put_line('Just IN OUT: ' || TO_CHAR((t2 - t1)/100.0));
   dbms_output.put_line('With NOCOPY: ' || TO_CHAR((t3 - t2))/100.0);
END;
/
```

Restrictions on NOCOPY

The use of NOCOPY increases the likelihood of parameter aliasing. For more information, see "Understanding Subprogram Parameter Aliasing".

Remember, NOCOPY is a hint, not a directive. In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method; no error is generated:

- The actual parameter is an element of an associative array. This restriction does not apply if the parameter is an entire associative array.
- The actual parameter is constrained, such as by scale or NOT NULL. This restriction does not apply to size-constrained character strings. This restriction does not extend to constrained elements or attributes of composite types.
- The actual and formal parameters are records, one or both records were declared using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit datatype conversion.
- The subprogram is called through a database link or as an external procedure.

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. Procedures compiled this way work in all server environments, such as the shared server configuration (formerly known as multi-threaded server) and Oracle Real Application Clusters.

Before You Begin

If you are a first-time user of native PL/SQL compilation, try it first with a test database, before proceeding to a production environment.

Always back up your database before configuring the database for PL/SQL native compilation. If you find that the performance benefit is outweighed by extra compilation time, it might be faster to restore from a backup than to recompile everything in interpreted mode.

Some of the setup steps require DBA authority. You must change the values of some initialization parameters, and create a new directory on the database server, preferably near the data files for the instance. The database server also needs a C compiler; on a cluster, the compiler is needed on each node. Even if you can test out these steps yourself on a development machine, you will generally need to consult with a DBA and enlist their help to use native compilation on a production server.

Contact your system administrator to ensure that you have the required C compiler on your operating system, and find the path for its location. Use a text editor such as vi to open the file `$ORACLE_HOME/plsql/spnc_commands`, and make sure the command templates are correct. Generally, you should not need to make any changes here, just confirm that the setup is correct.

Determining Whether to Use PL/SQL Native Compilation

PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations. Examples of such operations are data warehouse applications, and applications with extensive server-side transformations of data for display. In such cases, expect speed increases of up to 30%.

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. You can test to see how much performance gain you can get by enabling PL/SQL native compilation.

It takes longer to compile program units with native compilation than to use the default interpreted mode. You might turn off native compilation during the busiest parts of the development cycle, where code is being frequently recompiled.

When you have decided that you will have significant performance gains in database operations using PL/SQL native compilation, Oracle Corporation recommends that you compile the whole database using the `NATIVE` setting. Compiling all the PL/SQL code in the database means you see the speedup in your own code, and in calls to all the built-in PL/SQL packages.

How PL/SQL Native Compilation Works

If you do not use native compilation, each PL/SQL program unit is compiled into an intermediate form, machine-readable code (m-code). The m-code is stored in the database dictionary and interpreted at run time.

With PL/SQL native compilation, the PL/SQL statements are turned into C code that bypasses all the runtime interpretation, giving faster runtime performance.

PL/SQL uses the command file `$ORACLE_HOME/plsql/spnc_commands`, and the supported operating system C compiler and linker, to compile and link the resulting C code into shared libraries. The shared libraries are stored inside the data dictionary, so that they can be backed up automatically and are protected from being deleted. These shared library files are copied to the filesystem and are loaded and run when the PL/SQL subprogram is invoked. If the files are deleted from the filesystem while the database is shut down, or if you change the directory that holds the libraries, they are extracted again automatically.

Although PL/SQL program units that just call SQL statements might see little or no speedup, natively compiled PL/SQL is always at least as fast as the corresponding interpreted code. The compiled code makes the same library calls as the interpreted code would, so its behavior is exactly the same.

Format of the spnc_commands File

The spnc_commands file, in the `$ORACLE_HOME/plsql` directory, contains the templates for the commands to compile and link each program. Some special names such as `%(src)` are predefined, and are replaced by the corresponding filename. The variable `$(ORACLE_HOME)` is replaced by the location of the Oracle home directory. You can include comment lines, starting with a # character. The file contains comments that explain all the special notation.

The `spnc_commands` file contains a predefined path for the C compiler, depending on the particular operating system. (One specific compiler is supported on each operating system.) In most cases, you should not need to change this path, but you might if you the system administrator has installed it in another location.

System-Level Initialization Parameters for PL/SQL Native Compilation

The following table lists the initialization parameters you must set before using PL/SQL native compilation. They can be set only at the system level, not by an `ALTER SESSION` command. You cannot use variables such as `ORACLE_HOME` in the values; use the full path instead.

> **Note:** The examples in this section for setting system parameters for PL/SQL native compilation assume a system using a server parameter file (SPFILE).
>
> If you use a text initialization parameter file (PFILE, or `init`*`sid`*`.ora`), ensure that you change parameters in your initialization parameter file, as indicated in the following table.

Parameter	Characteristics
PLSQL_NATIVE_LIBRARY_DIR	The full path and directory name used to store the shared libraries that contain natively compiled PL/SQL code. In accordance with optimal flexible architecture (OFA) rules, Oracle Corporation recommends that you create the shared library directory as a subdirectory where the data files are located. For security reasons, only the users `oracle` and `root` should have write privileges for this directory.

Parameter	Characteristics
PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT	The number of subdirectories in the directory specified by the parameter PLSQL_NATIVE_LIBRARY_DIR. Optional; use if the number of natively compiled program units exceeds 15000. If you need to set this option, refer to the section "Setting Up PL/SQL Native Library Subdirectories" on page 11-27.

Session-Level Initialization Parameter for PL/SQL Native Compilation

The parameter PLSQL_CODE_TYPE determines whether PL/SQL code is natively compiled or interpreted. The default setting is INTERPRETED. To enable PL/SQL native compilation, set the value of PLSQL_CODE_TYPE to NATIVE.

If you compile the whole database as NATIVE, Oracle Corporation recommends that you set PLSQL_CODE_TYPE at the system level.

Use the following syntax to set this parameter:

```
alter session set plsql_code_type='NATIVE';
alter session set plsql_code_type='INTERPRETED';
alter system set plsql_code_type='NATIVE';
alter system set plsql_code_type='INTERPRETED';
```

> **See Also:** *Oracle Database Reference* for complete details about the initialization parameters and data dictionary views.

Setting Up and Using PL/SQL Native Compilation

To speed up one or more subprograms through native compilation:

1. Set up the PLSQL_NATIVE_LIBRARY_DIR initialization parameter, and optionally the PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT initialization parameter, as described above.
2. Use the ALTER SYSTEM or ALTER SESSION command, or update your initialization file, to set the parameter PLSQL_CODE_TYPE to the value NATIVE.
3. Compile one or more subprograms, using one of these methods:
   - Use CREATE OR REPLACE to create or recompile the subprogram.
   - Use the ALTER PROCEDURE, ALTER FUNCTION, or ALTER PACKAGE command with the COMPILE option to recompile the subprogram or the entire package. (You can also use the clause PLSQL_CODE_TYPE = NATIVE with the ALTER statements to affect specific subprograms without changing the initialization parameter for the whole session.)
   - Drop the subprogram and create it again.
   - Run one of the SQL*Plus scripts that creates a set of Oracle-supplied packages.
   - Create a database using a preconfigured initialization file with PLSQL_CODE_TYPE=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_PLSQL_OBJECT_SETTINGS,

DBA_PLSQL_OBJECT_SETTINGS, and ALL_PLSQL_OBJECT_SETTINGS. For example, to check the status of the procedure MY_PROC, you could enter:

```
CREATE OR REPLACE PROCEDURE my_proc AS BEGIN NULL; END;
/
SELECT plsql_code_type FROM user_plsql_object_settings WHERE name = 'MY_PROC';
DROP PROCEDURE my_proc;
```

The CODE_TYPE 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 interpreted, all compiled for native execution, or a mixture of both.

Dependencies, Invalidation and Revalidation

This recompilation happens automatically invalidated, such as when a table that it depends on is re-created.

If an object on which a natively compiled PL/SQL subprogram depends changes, the subprogram is invalidated. The next time the same subprogram is called, the database recompiles the subprogram automatically. Because the PLSQL_CODE_TYPE setting is stored inside the library unit for each subprogram, the automatic recompilation uses this stored setting for code type.

The stored settings are only used when recompiling as part of revalidation. If a PL/SQL subprogram is explicitly compiled through the SQL commands "`create or replace`" or "`alter...compile`", the current session setting is used.

The generated shared libraries are stored in the database, in the SYSTEM tablespace. The first time a natively compiled procedure is executed, the corresponding shared library is copied from the database to the directory specified by the initialization parameter PLSQL_NATIVE_LIBRARY_DIR.

Setting Up Databases for PL/SQL Native Compilation

Use the procedures in this section to set up an entire database for PL/SQL native compilation. The performance benefits apply to all the built-in PL/SQL packages, which are used for many database operations.

Creating a New Database for PL/SQL Native Compilation

If you use Database Configuration Assistant, use it to set the initialization parameters required for PL/SQL native compilation, as described in the preceding section, "System-Level Initialization Parameters for PL/SQL Native Compilation".

To find the supported C compiler on your operating system. refer to the table "Precompilers and Tools Restrictions and Requirements" in the installation guide for your operating system. Determine from your system administrator where it is located on your system. You will need to check that this path is correct in the spnc_commands file.

Determine if you have so many PL/SQL program units that you need to set the initialization parameter PLSQL_NATIVE_DIR_SUBDIR_COUNT, and create PL/SQL native library subdirectories if necessary. By default, PL/SQL program units are kept in one directory. If the number of program units exceeds 15,000, the operating system begins to impose performance limits. To work around this problem, Oracle

Corporation recommends that you spread the PL/SQL program units among subdirectories.

If you have set up a test database, use this SQL query to determine how many PL/SQL program units you are using:

```
select count (*) from DBA_PLSQL_OBJECTS;
```

If the count returned by this query is greater than 15,000, complete the procedure described in the section "Setting Up PL/SQL Native Library Subdirectories".

Modifying an Existing Database for PL/SQL Native Compilation

To natively compile an existing database, complete the following procedure:

1. Contact your system administrator to ensure that you have the required C compiler on your operating system, and find the path for its location. Use a text editor such as vi to open the file `spnc_commands`, and make sure the command templates are correct.
2. As the `oracle` user, create the PL/SQL native library directory for each Oracle database.

   Note: You must set up PL/SQL libraries for each Oracle database. Shared libraries (`.so` and `.dll` files) are logically connected to the database. They cannot be shared between databases. If you set up PL/SQL libraries to be shared, the databases will be corrupted.

   Create a directory in a secure place, in accordance with OFA rules, to prevent `.so` and `.dll` files from unauthorized access.

   In addition, ensure that the compiler executables used for PL/SQL native compilation are writable only by a properly secured user.

   The original copies of the shared libraries are stored inside the database, so they are backed up automatically with the database.

   ---

3. Using SQL, set the initialization parameter `PLSQL_NATIVE_LIBRARY_DIR` to the full path to the PL/SQL native library.

   For example, if the path to the PL/SQL native library directory is `/oracle/oradata/mydb/natlib`, enter the following:

   ```
   alter system set plsql_native_library_dir='/oracle/oradata/mydb/natlib'
   ```

4. Determine if you need to set the initialization parameter `PLSQL_NATIVE_DIR_SUBDIR_COUNT`, and create PL/SQL native library subdirectories if necessary.

   By default, PL/SQL program units are kept in one directory. However, if the number of program units exceeds 15000, then the operating system begins to impose performance limits. To work around this problem, Oracle Corporation recommends that you spread the PL/SQL program units in subdirectories.

   If you have an existing database that you will migrate to the new installation, or if you have set up a test database, use the following SQL query to determine how many PL/SQL program units you are using:

   ```
   select count (*) from DBA_PLSQL_OBJECTS;
   ```

If the count returned by this query is greater than 15,000, complete the procedure described in the following section, "Setting Up PL/SQL Native Library Subdirectories".

5. Set the remaining required initialization parameters as listed in the table in the preceding section "System-Level Initialization Parameters for PL/SQL Native Compilation".

6. Create the following stored procedure to confirm that PL/SQL native compilation is enabled:

```
CREATE OR REPLACE PROCEDURE Hello AS
BEGIN
   dbms_output.put_line ( 'This output is from a natively compiled procedure.'
);
END Hello;
/
```

7. Run the stored procedure:

```
CALL Hello();
```

   If the program does not return the expected output, contact Oracle Support for assistance. (Remember to `SET SERVEROUTPUT ON` in SQL*Plus before running the procedure.)

8. Recompile all the PL/SQL subprograms in the database. The script `$ORACLE_HOME/admin/utlirp.sql` is typically used here.

Setting Up PL/SQL Native Library Subdirectories

If you need to set up PL/SQL native library subdirectories, use the following procedure:

1. Create subdirectories sequentially in the form of d0, d1, d2, d3...dx, where x is the total number of directories. Oracle Corporation recommends that you use a script for this task. For example, you might run a PL/SQL block like the following, save its output to a file, then run that file as a shell script:

```
BEGIN
  FOR j IN 0..999
  LOOP
    dbms_output.put_line ( 'mkdir d' || TO_CHAR(j) );
  END LOOP;
END;
/
```

2. Set the initialization parameter `PLSQL_NATIVE_DIR_COUNT` to the number of subdirectories you have created. For example, if you created 1000 subdirectories, enter the following SQL statement in SQL*Plus:

```
alter system set plsql_native_library_subdir_count=1000;
```

Example 11–8 Compiling a PL/SQL Procedure for Native Execution

```
alter session set plsql_code_type='NATIVE';
CREATE OR REPLACE PROCEDURE hello_native
AS
BEGIN
  dbms_output.put_line('Hello world.');
  dbms_output.put_line('Today is ' || TO_CHAR(SYSDATE) || '.');
```

```
END;
/
select plsql_code_type from user_plsql_object_settings
   where name = 'HELLO_NATIVE';
alter session set plsql_code_type='INTERPRETED';
```

The procedure is immediately available to call, and runs as a shared library directly within the Oracle process. If any errors occur during compilation, you can see them using the `USER_ERRORS` view or the `SHOW ERRORS` command in SQL*Plus.

Limitations of Native Compilation

- Debugging tools for PL/SQL do not handle procedures compiled for native execution.
- When many procedures and packages (typically, over 15000) are compiled for native execution, the large number of shared objects in a single directory might affect system performance. See "Setting Up PL/SQL Native Library Subdirectories" on page 11-27 for a workaround.

Real Application Clusters and PL/SQL Native Compilation

Because any node might need to compile a PL/SQL subprogram, each node in the cluster needs a C compiler and correct settings and paths in the `$ORACLE_HOME/plsql/spnc_commands` file.

When you use PLSQL native compilation in a Real Application Clusters (RAC) environment, the original copies of the shared library files are stored in the databases, and these files are automatically propagated to all nodes in the cluster. You do not need to do any copying of libraries for this feature to work.

The reason for using a server parameter file (SPFILE) in the examples in this section, is to make sure that all nodes of a RAC cluster use the same settings for the parameters that control PL/SQL native compilation.

Setting Up Transformation Pipelines with Table Functions

This section describes how to chain together special kinds of functions known as table functions. These functions are used in situations such as data warehousing to apply multiple transformations to data.

Major topics covered are:

- Overview of Table Functions
- Writing a Pipelined Table Function

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 or assigned to a PL/SQL collection variable. You can use a table function like the name of a database table, in the `FROM` clause of a query, or like a column name in the `SELECT` list of a query.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type (such as a `VARRAY` or a PL/SQL table) or a `REF CURSOR`.

Execution of a table function can be parallelized, and returned rows can be streamed directly to the next process without intermediate staging. Rows from a collection returned by a table function can also be **pipelined**—that is, iteratively returned as they

are produced instead of in a batch after all processing of the table function's input is completed.

Streaming, pipelining, and parallel execution of table functions can improve performance:

- By enabling 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.

Example 11–9 Example: Querying a Table Function

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:

```
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:
CREATE TABLE Catalogs ( name VARCHAR2(30), cat CLOB );
```

Function `GetBooks` is defined as follows:

```
CREATE FUNCTION GetBooks(a CLOB) RETURN BookSet_t;
/
```

The query below returns all the catalogs and their corresponding book listings.

```
SELECT c.name, Book.name, Book.author, Book.abstract
  FROM Catalogs c, TABLE(GetBooks(c.cat)) Book;
```

Example 11–10 Example: Assigning the Result of a Table Function

The following example shows how you can assign the result of a table function to a PL/SQL collection variable. Because the table function is called from the `SELECT` list of the query, you do not need the `TABLE` keyword.

```
CREATE TYPE numset_t AS TABLE OF NUMBER;
/

CREATE FUNCTION f1(x number) RETURN numset_t PIPELINED IS
BEGIN
  FOR i IN 1..x LOOP
    PIPE ROW(i);
```

```
  END LOOP;
  RETURN;
END;
/

-- pipelined function in from clause
select * from table(f1(3));
```

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.

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 "Returning Results from Table Functions" on page 11-31 for the function bodies.

```
-- 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 returns 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.)

```
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:

```
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.

The `PIPE ROW` statement may be used only in the body of pipelined table functions; an error is raised if it is used anywhere else. The `PIPE ROW` statement can be omitted for a pipelined table function that returns no rows.

A pipelined table function must have a `RETURN` statement that does not return a value. The `RETURN` statement transfers the control back to the consumer and ensures that the next fetch gets a `NO_DATA_FOUND` exception.

Because table functions pass control back and forth to a calling routine as rows areproduced, there is a restriction on combining table functions and `PRAGMA AUTONOMOUS_TRANSACTION`. If a table function is part of an autonomous transaction, it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement, to avoid an error in the calling subprogram.

Oracle has three special SQL datatypes that enable you to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these three special types to create **anonymous** (that is, unnamed) types, including anonymous collection types. The types are `SYS.ANYTYPE`, `SYS.ANYDATA`, and `SYS.ANYDATASET`. The `SYS.ANYDATA` type can be useful in some situations as a return value from table functions.

> **See Also:** *PL/SQL Packages and Types Reference* for information about the interfaces to the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types and about the `DBMS_TYPES` package for use with these types.

Pipelining Data Between PL/SQL Table Functions

With serial execution, results are pipelined from one PL/SQL table function to another using an approach similar to co-routine execution. For example, the following statement pipelines results from function `g` to function `f`:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));
```

Parallel execution works similarly except that each function executes in a different process (or set of processes).

Querying Table Functions

Pipelined table functions are used in the `FROM` clause of `SELECT` statements. The result rows are retrieved by Oracle iteratively from the table function implementation. For example:

```
SELECT x.Ticker, x.Price
FROM TABLE(StockPivot(  CURSOR(SELECT * FROM StockTable))) x
WHERE x.PriceType='C';
```

Note: A table function returns a collection. In some cases, such as when the top-level query uses `SELECT *` and the query refers to a PL/SQL variable or a bind variable, you may need a `CAST` operator around the table function to specify the exact return type.

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,

```
SELECT * FROM TABLE(f(...)) t1, TABLE(f(...)) t2
  WHERE t1.id = t2.id;

SELECT * FROM TABLE(f());
SELECT * FROM TABLE(f());
```

If the function always produces the same result value for each combination of values passed in, you can declare the function `DETERMINISTIC`, and Oracle automatically buffers rows for it. If the function is not really deterministic, results are unpredictable.

Fetching from the Results of Table Functions

PL/SQL cursors and ref cursors can be defined for queries over table functions. For example:

```
OPEN c FOR SELECT * FROM TABLE(f(...));
```

Cursors over table functions have the same fetch semantics as ordinary cursors. `REF CURSOR` assignments based on table functions do not have any special semantics.

However, the SQL optimizer will not optimize across PL/SQL statements. For example:

```
DECLARE
    r SYS_REFCURSOR;
BEGIN
    OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
    SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
END;
/
```

does not execute as well as:

```
SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
```

This is so even ignoring the overhead associated with executing two SQL statements and assuming that the results can be pipelined between the two statements.

Passing Data with Cursor Variables

You can pass a set of rows to a PL/SQL function in a `REF CURSOR` parameter. For example, this function is declared to accept an argument of the predefined weakly typed `REF CURSOR` type `SYS_REFCURSOR`:

```
FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;
```

Results of a subquery can be passed to a function directly:

```
SELECT * FROM TABLE(f(CURSOR(SELECT 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.

Example 11–11 Example: Using Multiple REF CURSOR Input Variables

PL/SQL functions can accept multiple REF CURSOR input variables:

```
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 employee_id FROM tab),
  CURSOR(SELECT * FROM employees));
```

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(...)))));
```

Example 11–12 Example: 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:

```
DECLARE
  r SYS_REFCURSOR;
  rec ...;
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(...));
  -- Must return a single row result set.
  SELECT * INTO rec FROM TABLE(g(r));
END;
/
```

In this case, the table function closes the cursor when it completes, so your program should not explicitly try to close the cursor.

Example 11–13 Example: Using a Pipelined Table Function as an Aggregate Function

A table function can compute aggregate results using the input ref cursor. The following example computes a weighted average by iterating over a set of input rows.

```
DROP TABLE gradereport;
CREATE TABLE gradereport (student VARCHAR2(30), subject VARCHAR2(30), weight
NUMBER, grade NUMBER);

INSERT INTO gradereport  VALUES('Mark', 'Physics', 4, 4);
INSERT INTO  gradereport VALUES('Mark','Chemistry', 4,3);
INSERT INTO gradereport VALUES('Mark','Maths', 3,3);
INSERT INTO gradereport VALUES('Mark','Economics', 3,4);
```

```
CREATE OR replace TYPE gpa AS TABLE OF NUMBER;
/

CREATE OR replace FUNCTION weighted_average(input_values
sys_refcursor)
RETURN gpa PIPELINED IS
  grade NUMBER;
  total NUMBER := 0;
  total_weight NUMBER := 0;
  weight NUMBER := 0;
BEGIN
-- The function accepts a ref cursor and loops through all the input rows.
  LOOP
     FETCH input_values INTO weight, grade;
     EXIT WHEN input_values%NOTFOUND;
-- Accumulate the weighted average.
     total_weight := total_weight + weight;
     total := total + grade*weight;
  END LOOP;
  PIPE ROW (total / total_weight);
-- The function returns a single result.
  RETURN;
END;
/
show errors;

-- The result comes back as a nested table with a single row.
-- COLUMN_VALUE is a keyword that returns the contents of a nested table.
select weighted_result.column_value from
   table( weighted_average( cursor(select weight,grade from gradereport) ) )
weighted_result;
```

Performing DML Operations Inside Table Functions

To execute DML statements, declare a table function with the AUTONOMOUS_TRANSACTION pragma, which causes the function to execute in a new transaction not shared by other processes:

```
CREATE FUNCTION f(p SYS_REFCURSOR) return CollType PIPELINED IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN NULL; END;
/
```

During parallel execution, each instance of the table function creates an independent transaction.

Performing DML Operations on 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;
INSERT INTO f(...) VALUES ('any', 'thing');
```

However, you can create a view over a table function and use INSTEAD OF triggers to update it. For example:

```
CREATE VIEW BookTable AS
  SELECT x.Name, x.Author
```

```
  FROM TABLE(GetBooks('data.txt')) x;
```

The following `INSTEAD OF` trigger is fired when the user inserts a row into the `BookTable` view:

```
CREATE TRIGGER BookTable_insert
INSTEAD OF INSERT ON BookTable
REFERENCING NEW AS n
FOR EACH ROW
BEGIN
  ...
END;
/
INSERT INTO BookTable VALUES (...);
```

`INSTEAD OF` triggers can be defined for all DML operations on a view built on a table function.

Handling Exceptions in Table Functions

Exception handling in table functions works just as it does with regular functions.

Some languages, such as C and Java, provide a mechanism for user-supplied exception handling. If an exception raised within a table function is handled, the table function executes the exception handler and continues processing. Exiting the exception handler takes control to the enclosing scope. If the exception is cleared, execution proceeds normally.

An unhandled exception in a table function causes the parent transaction to roll back.

12

Using PL/SQL Object Types

... It next will be right
To describe each particular batch:
Distinguishing those that have feathers, and bite,
From those that have whiskers, and scratch. —Lewis Carroll

Object-oriented programming is especially suited for building reusable components and complex applications. In PL/SQL, object-oriented programming is based on object types. They let you model real-world objects, separate interfaces and implementation details, and store object-oriented data persistently in the database. You might find object types helpful when writing programs that interoperate with Java or other object-oriented languages.

This chapter contains these topics:

Overview of PL/SQL Object Types

Before reading this chapter, you should be familiar with some backgroup topics:

Object-oriented programming, including the idea of abstraction.

The ideas of attributes and methods. In other languages, these are part of classes. In SQL and PL/SQL, they are part of object types.

The SQL statement CREATE TYPE.

The best place to read about all of Oracle's object-oriented features is... The remainder of this chapter focuses on issues that are specific to PL/SQL.

What Is an Object Type?

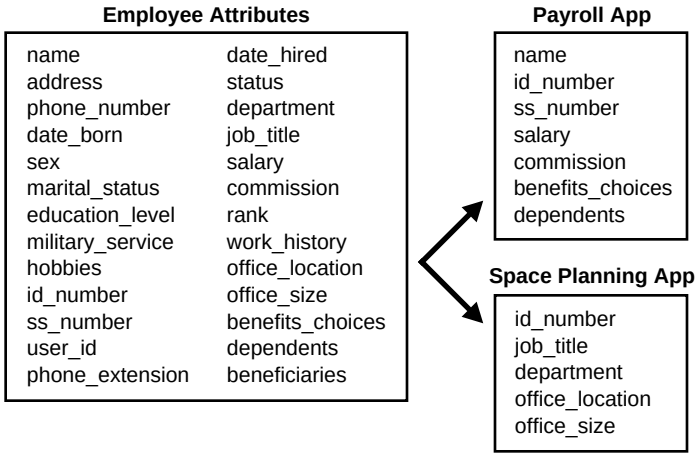
An **object type** is a user-defined composite datatype representing a data structure and functions and procedures to manipulate the data. With scalar datatypes, each variable holds a single value. With collections, all the elements have the same type. Only object types let you associate code with the data.

The variables within the data structure are called **attributes**. The functions and procedures of the object type are called **methods**.

We usually think of an object as having attributes and actions. For example, a baby has the attributes gender, age, and weight, and the actions eat, drink, and sleep. Object types let you represent such real-world behavior in 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 the attributes and behaviors the object needs in the application environment.

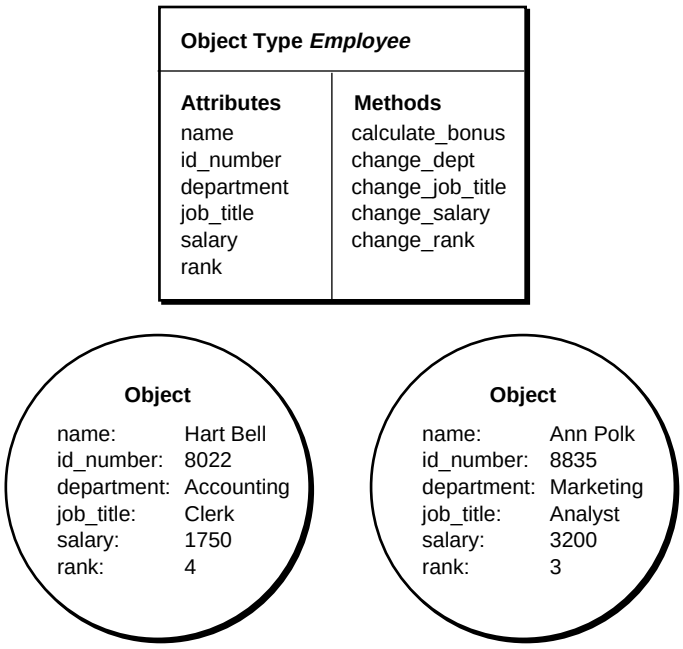
Figure 12–1 Each Application Uses a Subset of Object Attributes



Although the attributes are public (visible to client programs), well-behaved programs manipulate the data only through methodsthat you provide, not by assigning or reading values directly. Because the methods can do extra checking, the data is kept in a proper state.

At run time, you create **instances** of an abstract type, real objects with filled-in attributes.

Figure 12–2 Object Type and Objects (Instances) of that Type



Why Use Object Types?

Object types let you break down a large system into logical entities. This lets you create software components that are modular, maintainable, and reusable across projects and teams.

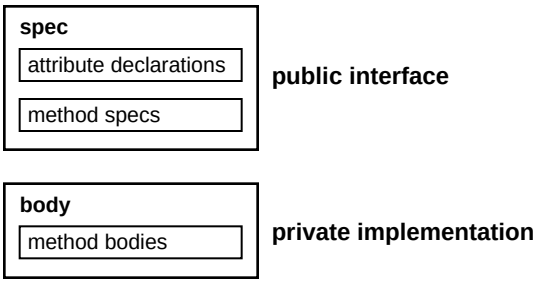
By associating code with data, object types move maintenance code out of SQL scripts and PL/SQL blocks into methods. Instead of writing a long procedure that does different things based on some parameter value, you can define different object types and make each operate slightly differently. By declaring an object of the correct type, you ensure that it can only perform the operations for that type.

Object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. Object types map directly into classes defined in object-oriented languages such as Java and C++.

Structure of an Object Type

Like a package, an object type has a specification and a body (refer to Figure 12–3). The **specification** (or **spec** for short) defines the programming interface; it declares a set of attributes along with the operations (methods) to manipulate the data. The **body** defines the code for the methods.

Figure 12–3 Object Type Structure



All the information a program needs to use the methods is in the spec. You can change the body without changing the spec, and without affecting client programs.

In an object type spec, all attributes must be declared before any methods. 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).

The following example defines an object type for complex numbers, with a real part, an imaginary part, and arithmetic operations.

```
CREATE TYPE Complex AS OBJECT (
   rpart REAL,  -- "real" attribute
   ipart REAL,  -- "imaginary" attribute
   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`
- `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 12-5.) The same restrictions apply to return types.

What Languages can I Use for Methods of Object Types?

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 *Oracle Database Java Developer's Guide*. To learn how to write C call specs, see *Oracle Database Application Developer's Guide - Fundamentals*.

How Object Types Handle the SELF Parameter

`MEMBER` methods accept a built-in parameter named `SELF`, which is an instance of the object type. It is always the first parameter passed to a `MEMBER` method. If you do not declare it, it declared automatically.

For example, the `transform` method declares `SELF` as an `IN OUT` parameter:

```
CREATE TYPE Complex AS OBJECT (
   MEMBER FUNCTION transform (SELF IN OUT Complex) ...
```

You must specify the same datatype for `SELF` as the original object.

In `MEMBER` functions, if `SELF` is not declared, its parameter mode defaults to `IN`.

In `MEMBER` procedures, if `SELF` is not declared, its parameter mode defaults to `IN OUT`.

You cannot specify the `OUT` parameter mode for `SELF`.

`STATIC` methods cannot accept or reference `SELF`.

In the method body, `SELF` denotes the object whose method was invoked. You can refer to `SELF.attribute_name` or `SELF.member_name`, to make clear that you are referring to that object rather than something in a supertype. As the following example shows, methods can reference the attributes of `SELF` without a qualifier:

```
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);
   ELSE ans := gcd(y, x MOD y);
   END IF;
   RETURN ans;
END;

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. You can use the same name for different methods if their formal parameters are different enough to tell apart. When you call one of the methods, PL/SQL finds it by comparing the 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 whose formal parameters differ only in their mode. You cannot overload two member functions that differ only in return type. For more information, see "Overloading Subprogram Names" on page 8-9.

MAP and ORDER Methods

Instances of an object type have no predefined order. To put them in order for comparisons or sorting, 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:

```
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;
```

PL/SQL uses the map method 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:

```
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 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 match() automatically. The method returns a negative number, zero, or a positive number signifying that SELF is 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. 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, a function with the same name as the object type that initializes and returns a new instance of that object type.

Oracle generates a default constructor for every object type, with formal parameters that match the order, names, and datatypes of the object attributes.

You can define your own constructor methods, either overriding a system-defined constructor, or defining a new function with a different signature.

PL/SQL never calls a constructor implicitly. You must call it explicitly.

For more information, see "Defining Object Constructors" on page 12-13.

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:

```
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);
ALTER TYPE mytype ADD ATTRIBUTE (attr3 NUMBER),
DROP ATTRIBUTE attr2,
ADD ATTRIBUTE attr4 NUMBER CASCADE;
```

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 *Oracle Database SQL Reference*. For guidelines about using type evolution in your applications, and options for changing other types and data that rely on those types, see *Oracle Database Application Developer's Guide - Object-Relational Features*.

Defining Object Types

An object type can represent any real-world entity. For example, an object type can represent a student, bank account, computer screen, rational number, or data structure such as a queue, stack, or list. This section gives several complete examples, which teach you a lot about the design of object types and prepare you to start writing your own.

Currently, you cannot define object types in a PL/SQL block, subprogram, or package. You can define them interactively in SQL*Plus using the SQL statement CREATE TYPE.

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 to return only those objects that are of a specified subtype.

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 *Oracle Database 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_type 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;
```

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 run time, 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.

```
DECLARE
   r Rational;
BEGIN
   r := Rational(6, 8);
   dbms_output.put_line(r.num); -- prints 6
```

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:

```
DECLARE
   ...
   PROCEDURE open_acct (new_acct IN OUT Account) IS ...
```

In the following example, you use object type `Account` to specify the return type of a function:

```
DECLARE
   ...
   FUNCTION get_acct (acct_id IN INTEGER) RETURN Account IS ...
```

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:

```
DECLARE
   r Rational;  -- r becomes atomically null
BEGIN
   r := Rational(2,3);  -- r becomes 2/3
```

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:

```
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 Object Attributes

You refer to an attribute by name. To access or change the value of an attribute, you use dot notation:

```
DECLARE
   r Rational := Rational(NULL, NULL);
   numerator   INTEGER;
   denominator INTEGER;
BEGIN
   denominator := r.den; -- Read value of attribute
   r.num := numerator;   -- Assign value to attribute
```

Attribute names can be chained, which lets you access the attributes of a nested object type. For example, suppose you define object types Address and Student, as follows:

```
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),
   ...
);
```

The Address attribute is an object type that has a zip_code attribute. If s is a Student object, you access the value of its zip_code attribute as follows:

```
s.home_address.zip_code
```

Defining Object Constructors

By default, you do not need to define a constructor for an object type. The system supplies a default constructor that accepts a parameter corresponding to each attribute.

You might also want to define your own constructor:

- To supply default values for some attributes. You can ensure the values are correct instead of relying on the caller to supply every attribute value.
- To avoid many special-purpose procedures that just initialize different parts of an object.
- To avoid code changes in applications that call the constructor, when new attributes are added to the type. The constructor might need some new code, for

example to set the attribute to null, but its signature could remain the same so that existing calls to the constructor would continue to work.

For example:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(
-- The type has 3 attributes.
  length NUMBER,
  width NUMBER,
  area NUMBER,
-- Define a constructor that has only 2 parameters.
  CONSTRUCTOR FUNCTION rectangle(length NUMBER, width NUMBER)
    RETURN SELF AS RESULT
);
/

CREATE OR REPLACE TYPE BODY rectangle AS
  CONSTRUCTOR FUNCTION rectangle(length NUMBER, width NUMBER)
    RETURN SELF AS RESULT
  AS
  BEGIN
    SELF.length := length;
    SELF.width := width;
-- We compute the area rather than accepting it as a parameter.
    SELF.area := length * width;
    RETURN;
  END;
END;
/

DECLARE
  r1 rectangle;
  r2 rectangle;
BEGIN
-- We can still call the default constructor, with all 3 parameters.
  r1 := NEW rectangle(10,20,200);
-- But it is more robust to call our constructor, which computes
-- the AREA attribute. This guarantees that the initial value is OK.
  r2 := NEW rectangle(10,20);
END;
/
```

Calling Object 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. When you call the default constructor to fill in all attribute values, you must supply a parameter for every attribute; 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 Object 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;
BEGIN
   r := Rational(6, 8);
   r.normalize;
   dbms_output.put_line(r.num);  -- prints 3
END;
```

As the example below shows, you can chain method calls. Execution proceeds from left to right. First, member function `reciprocal()` is called, then member procedure `normalize()` is called.

```
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:

```
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 *`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 *dynamic method dispatch*.

Sharing Objects through the REF Modifier

It is inefficient to pass copies of large objects from subprogram to subprogram. It makes more sense to pass a pointer instead. A *ref* is a pointer to an object.

Sharing means that data is not replicated unnecessarily. When a shared object is updated, the change occurs in only one place, and any ref can retrieve the updated values instantly.

```
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;
```

By revising object type `Person`, you can model families, where several people 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. You can use refs as input or output variables in SQL data manipulation statements.

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 not allowed:

```
DECLARE
   p_ref    REF Person;
   phone_no VARCHAR2(15);
BEGIN
   phone_no := p_ref.phone_number;  -- not allowed
```

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 12-20.

Forward Type Definitions

You can refer only to schema objects that already exist. In the following example, the first CREATE TYPE statement is not allowed because it refers to object type Department, which does not yet exist:

```
CREATE TYPE Employee AS OBJECT (
   name VARCHAR2(20),
   dept REF Department,  -- not allowed
   ...
);

CREATE TYPE Department AS OBJECT (
   number  INTEGER,
   manager Employee,
   ...
);
```

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. The incomplete type Customer can be made available to other application developers for use in refs.

Manipulating Objects through SQL

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';
```

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';
```

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';
```

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';
```

The next example retrieves 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;
```

This example, updates 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 to dereference a pointer, and get the value to which it points. `DEREF` takes a reference to an object, and returns the value of that object. If the ref is dangling, `DEREF` returns a null object.

The following example dereferences a ref to a `Person` object. You can select from the dummy table `DUAL` because each object stored in an object table has a unique object identifier, which is part of every ref to that object.

```
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;
```

You can use `DEREF` in successive SQL statements to dereference refs:

```
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
   p1 := 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 by using a table alias. For example, the following syntax is valid:

```
table_alias.object_column.ref_attribute
table_alias.object_column.ref_attribute.attribute
table_alias.ref_column.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:

```
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.

```
DECLARE
   my_street VARCHAR2(25),
BEGIN
   SELECT p.home_address.street INTO my_street FROM persons p
      WHERE p.last_name = 'Lucas';
```

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`:

```
BEGIN
   INSERT INTO persons
      VALUES ('Jenifer', 'Lapidus', ...);
```

Alternatively, you can use the constructor for object type `Person` to insert an object into object table `persons`:

```
BEGIN
   INSERT INTO persons
      VALUES (Person('Albert', 'Brooker', ...));
```

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.

```
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;
```

To insert objects into an object table, you can use a subquery that returns objects of the same type. An example follows:

```
BEGIN
   INSERT INTO persons2
      SELECT VALUE(p) FROM persons p
         WHERE p.last_name LIKE '%Jones';
```

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 (
   dept_name 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, use the `WHERE` clause:

```
BEGIN
   DELETE FROM persons p
      WHERE p.home_address = '108 Palm Dr';
END;
```

13

PL/SQL Language Elements

Grammar, which knows how to control even kings. —Molière

This chapter is a quick reference guide to PL/SQL syntax and semantics. It shows you how commands, parameters, and other language elements are combined to form PL/SQL statements. It also provides usage notes and short examples.

This chapter contains these topics:

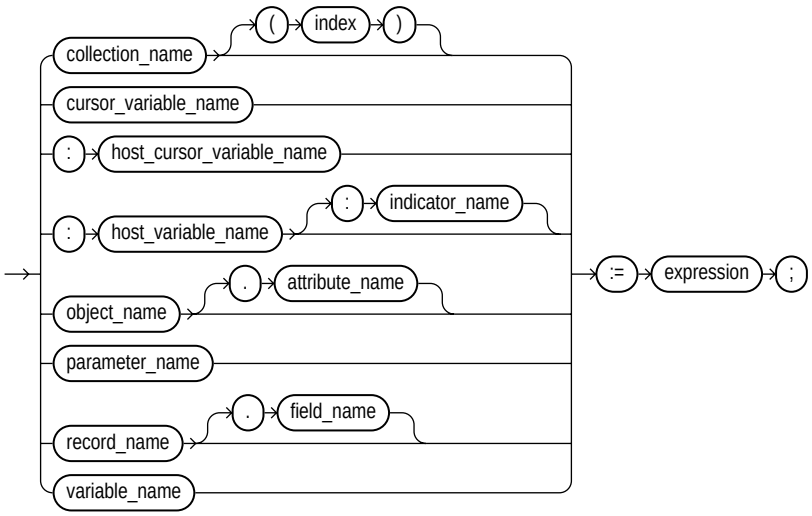
- IF Statement
- INSERT Statement
- Literals
- LOCK TABLE Statement
- LOOP Statements
- MERGE Statement
- NULL Statement
- Object Types
- OPEN Statement
- OPEN-FOR Statement
- OPEN-FOR-USING Statement
- Packages
- Procedures
- RAISE Statement
- Records
- RESTRICT_REFERENCES Pragma
- RETURN Statement
- ROLLBACK Statement
- %ROWTYPE Attribute
- SAVEPOINT Statement
- SCN_TO_TIMESTAMP Function
- SQLCODE Function
- SELECT INTO Statement
- SERIALLY_REUSABLE Pragma
- SET TRANSACTION Statement
- SQL Cursor
- SQLCODE Function
- SQLERRM Function
- TIMESTAMP_TO_SCN Function
- %TYPE Attribute
- UPDATE Statement

Assignment Statement

An assignment statement sets the current value of a variable, field, parameter, or element. The statement consists of an assignment target followed by the assignment operator and an expression. When the statement is executed, the expression is evaluated and the resulting value is stored in the target. For more information, see "Assigning Values to Variables" on page 2-16.

Syntax

assignment_statement



Keyword and Parameter Description

attribute_name

An attribute of an object type. The name must be unique within the object type (but can be reused in other object types). You cannot initialize an attribute in its declaration using the assignment operator or `DEFAULT` clause. Also, you cannot impose the `NOT NULL` constraint on an attribute.

collection_name

A nested table, index-by table, or varray previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of `expression`, see "Expressions" on page 13-52. When the assignment statement is executed, the expression is evaluated and the resulting value is stored in the assignment target. The value and target must have compatible datatypes.

field_name

A field in a user-defined or %ROWTYPE record.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind 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

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

A numeric expression that must return a value of type BINARY_INTEGER or a value implicitly convertible to that datatype.

indicator_name

An indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable "indicates" the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables let you detect nulls or truncated values in output host variables.

object_name

An instance of an object type previously declared within the current scope.

parameter_name

A formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears.

record_name

A user-defined or %ROWTYPE record previously declared within the current scope.

variable_name

A PL/SQL variable previously declared within the current scope.

Usage Notes

By default, unless a variable is initialized in its declaration, it is initialized to NULL every time a block or subprogram is entered. Always assign a value to a variable before using that variable in an expression.

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

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

You can assign the result of a comparison or other test to a Boolean variable.

You can assign the value of an expression to a specific field in a record.

You can assign values to all fields in a record at once. PL/SQL allows aggregate assignment between entire records if their declarations refer to the same cursor or table. The following example copies values from all the fields of one record to another:

```
DECLARE
   emp_rec1 emp%ROWTYPE;
   emp_rec2 emp%ROWTYPE;
   dept_rec dept%ROWTYPE;
BEGIN
   ...
   emp_rec1 := emp_rec2;
```

You can assign the value of an expression to a specific element in a collection, by subscripting the collection name.

Examples

```
DECLARE
   wages NUMBER; hours_worked NUMBER; hourly_salary NUMBER; bonus NUMBER;
   country VARCHAR2(128);
   counter NUMBER := 0;
   done BOOLEAN;
   emp_rec employees%ROWTYPE;
   TYPE commissions IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   comm_tab commissions;
BEGIN
   wages := (hours_worked * hourly_salary) + bonus;
   country := 'France';
   country := UPPER('Canada');
   done := (counter > 100);
   emp_rec.first_name := 'Antonio';
   comm_tab(5) := 20000 * 0.15;
END;
/
```

Related Topics

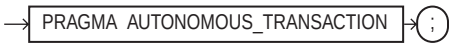
Constants and Variables, Expressions, SELECT INTO Statement

AUTONOMOUS_TRANSACTION Pragma

The `AUTONOMOUS_TRANSACTION` pragma changes the way a subprogram works within a transaction. A subprogram marked with this pragma can do SQL operations and commit or roll back those operations, without committing or rolling back the data in the main transaction. For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-35.

Syntax

autonomous_transaction_pragma



Keyword and Parameter Description

PRAGMA

Signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They pass information to the compiler.

Usage Notes

You can apply this pragma to:

- Top-level (not nested) anonymous PL/SQL blocks
- Local, standalone, and packaged functions and procedures
- Methods of a SQL object type
- Database triggers

You cannot apply this pragma to an entire package or an entire an object type. Instead, you can apply the pragma to each packaged subprogram or object method.

You can code the pragma anywhere in the declarative section. For readability, code the pragma at the top of the section.

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

Unlike regular triggers, autonomous triggers can contain transaction control statements such as `COMMIT` and `ROLLBACK`, and can issue DDL statements (such as `CREATE` and `DROP`) through the `EXECUTE IMMEDIATE` statement.

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. The changes also become visible to the main transaction when it resumes, but only if its isolation level is set to `READ COMMITTED` (the default). If you set the isolation level of the main transaction to `SERIALIZABLE`, changes made by its autonomous transactions are *not* visible to the main transaction when it resumes.

In the main transaction, rolling back to a savepoint located before the call to the autonomous subprogram does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

If an autonomous transaction attempts to access a resource held by the main transaction (which cannot resume until the autonomous routine exits), a deadlock can

occur. Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception. If the exception goes unhandled, or if the transaction ends because of some other unhandled exception, the transaction is rolled back.

Examples

The following example marks 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
      NULL;
   END;
END banking;
/

DROP PACKAGE banking;
```

The following example lets a trigger issue transaction control statements:

```
CREATE TABLE anniversaries AS
   SELECT DISTINCT TRUNC(hire_date) anniversary FROM employees;
ALTER TABLE anniversaries ADD PRIMARY KEY (anniversary);

CREATE TRIGGER anniversary_trigger
   BEFORE INSERT ON employees FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO anniversaries VALUES(TRUNC(:new.hire_date));
-- Only commits the preceding INSERT, not the INSERT that fired
-- the trigger.
   COMMIT;
   EXCEPTION
-- If someone else was hired on the same day, we get an exception
-- because of duplicate values. That's OK, no action needed.
      WHEN OTHERS THEN NULL;
END;
/

DROP TRIGGER anniversary_trigger;
DROP TABLE anniversaries;
```

Related Topics

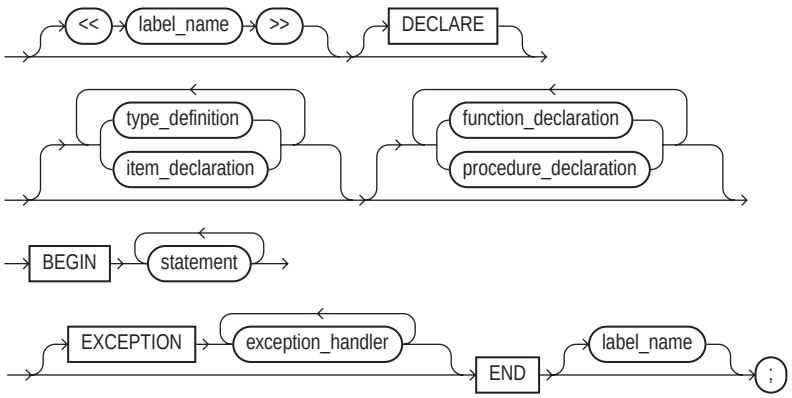
EXCEPTION_INIT Pragma, RESTRICT_REFERENCES Pragma, SERIALLY_REUSABLE Pragma

Blocks

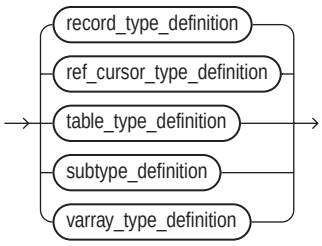
The basic program unit in PL/SQL is the block. A PL/SQL block is defined by the keywords `DECLARE`, `BEGIN`, `EXCEPTION`, and `END`. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required. You can nest a block within another block wherever you can place an executable statement. For more information, see "Block Structure" on page 1-4 and "Scope and Visibility of PL/SQL Identifiers" on page 2-14.

Syntax

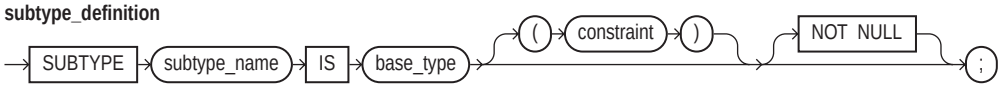
plsql_block



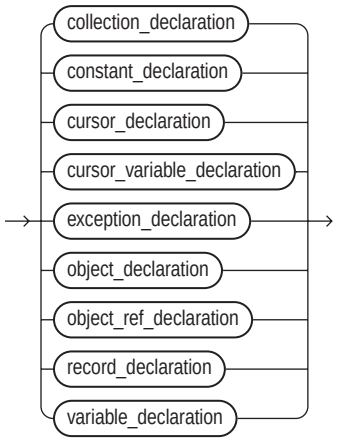
type_definition



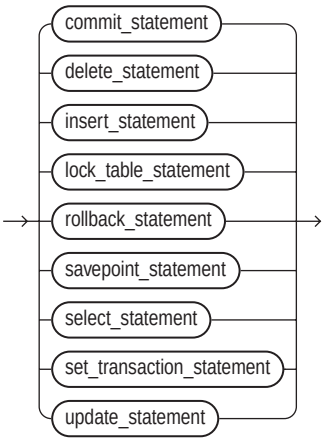
subtype_definition

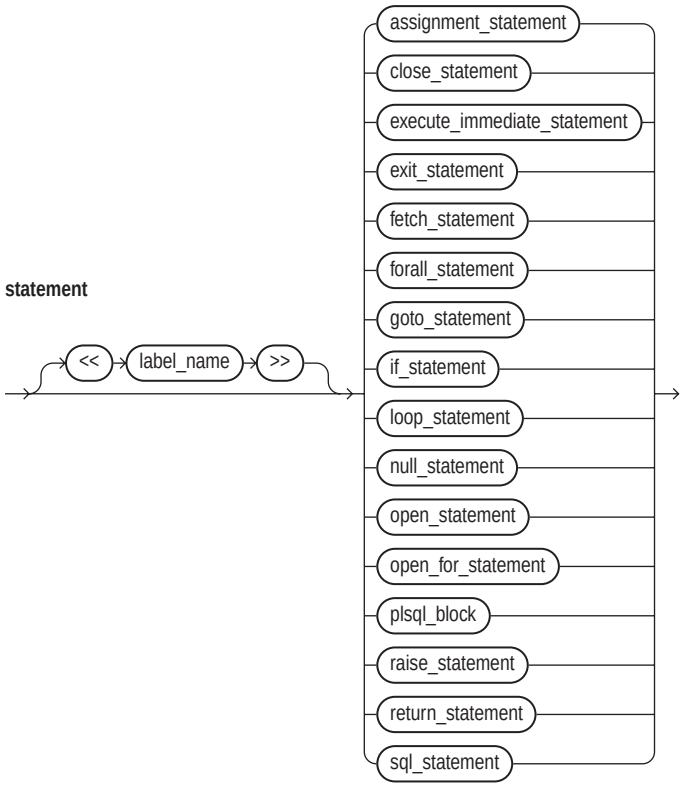


item_declaration



sql_statement





Keyword and Parameter Description

base_type

Any scalar or user-defined PL/SQL datatype specifier such as `CHAR`, `DATE`, or `RECORD`.

BEGIN

Signals the start of the executable part of a PL/SQL block, which contains executable statements. A PL/SQL block must contain at least one executable statement (even just the `NULL` statement).

collection_declaration

Declares a collection (index-by table, nested table, or varray). For the syntax of `collection_declaration`, see "Collections" on page 13-21.

constant_declaration

Declares a constant. For the syntax of `constant_declaration`, see "Constants and Variables" on page 13-28.

constraint

Applies only to datatypes that can be constrained such as `CHAR` and `NUMBER`. For character datatypes, this specifies a maximum size in bytes. For numeric datatypes, this specifies a maximum precision and scale.

cursor_declaration

Declares an explicit cursor. For the syntax of `cursor_declaration`, see "Cursors" on page 13-38.

cursor_variable_declaration

Declares a cursor variable. For the syntax of `cursor_variable_declaration`, see "Cursor Variables" on page 13-34.

DECLARE

Signals the start of the declarative part of a PL/SQL block, which contains local declarations. Items declared locally exist only within the current block and all its sub-blocks and are not visible to enclosing blocks. The declarative part of a PL/SQL block is optional. It is terminated implicitly by the keyword `BEGIN`, which introduces the executable part of the block.

PL/SQL does not allow forward references. You must declare an item before referencing it in any other statements. Also, you must declare subprograms at the end of a declarative section after all other program items.

END

Signals the end of a PL/SQL block. It must be the last keyword in a block. Remember, `END` does *not* signal the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.

EXCEPTION

Signals the start of the exception-handling part of a PL/SQL block. When an exception is raised, normal execution of the block stops and control transfers to the appropriate exception handler. After the exception handler completes, execution proceeds with the statement following the block.

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 10.

exception_declaration

Declares an exception. For the syntax of `exception_declaration`, see "Exceptions" on page 13-45.

exception_handler

Associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of `exception_handler`, see "Exceptions" on page 13-45.

function_declaration

Declares a function. For the syntax of `function_declaration`, see "Functions" on page 13-67.

label_name

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
   DECLARE
      x INTEGER;
   BEGIN
      IF x = outer.x THEN  -- refers to global x
         NULL;
      END IF;
   END;
END outer;
/
```

object_declaration

Declares an instance of an object type. For the syntax of `object_declaration`, see "Object Types" on page 13-86.

procedure_declaration

Declares a procedure. For the syntax of `procedure_declaration`, see "Procedures" on page 13-104.

record_declaration

Declares a user-defined record. For the syntax of `record_declaration`, see "Records" on page 13-110.

statement

An executable (not declarative) statement that. A sequence of statements can include procedural statements such as `RAISE`, SQL statements such as `UPDATE`, and PL/SQL blocks.

PL/SQL statements are free format. That is, they can continue from line to line if you do not split keywords, delimiters, or literals across lines. A semicolon (`;`) serves as the statement terminator.

subtype_name

A user-defined subtype that was defined using any scalar or user-defined PL/SQL datatype specifier such as `CHAR`, `DATE`, or `RECORD`.

variable_declaration

Declares a variable. For the syntax of `variable_declaration`, see "Constants and Variables" on page 13-28.

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 some variables, executes statements with calculations and function calls, and handles errors that might occur:

```
DECLARE
   numerator   NUMBER := 22;
   denominator NUMBER := 7;
   the_ratio   NUMBER;
BEGIN
   the_ratio := numerator/denominator;
   dbms_output.put_line('Ratio = ' || the_ratio);
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      dbms_output.put_line('Divide-by-zero error: can''t divide ' ||
         numerator || ' by ' || denominator);
   WHEN OTHERS THEN
      dbms_output.put_line('Unexpected error.');
END;
/
```

Related Topics

Constants and Variables, Exceptions, Functions, Procedures

CASE Statement

The CASE statement chooses from a sequence of conditions, and executes a corresponding statement. The CASE statement evaluates a single expression and compares it against several potential values, or evaluates multiple Boolean expressions and chooses the first one that is TRUE.

Syntax

searched_case_statement ::=

```
[ <<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

The WHEN clauses are executed in order.

Each WHEN clause is executed only once.

After a matching WHEN clause is found, subsequent WHEN clauses are not executed.

The statements in a WHEN clause can 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.

You can include CASE expressions inside SQL queries, for example instead of a call to the DECODE function or some other function that translates from one value to another.

Examples

The following example shows a simple CASE statement. Notice that you can use multiple statements after a WHEN clause, and that the expression in the WHEN clause can be a literal, variable, function call, or any other kind of expression.

```
DECLARE
  n number := 2;
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');
    WHEN 2+2 THEN
      dbms_output.put_line('n = 4');
    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 := 100;
  projected NUMBER := 30;
  needed NUMBER := 999;
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.');
  END CASE here;
  EXCEPTION
    WHEN CASE_NOT_FOUND THEN
      dbms_output.put_line('Somehow quantity is less than 0.');
END;
/
```

Related Topics

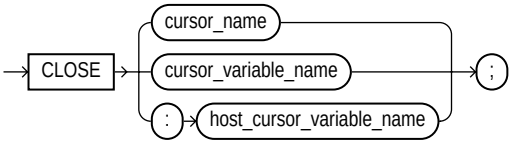
"Testing Conditions: IF and CASE Statements" on page 4-2, CASE Expressions on page 2-24, NULLIF and COALESCE expressions in *Oracle Database SQL Reference*

CLOSE Statement

The `CLOSE` statement indicates that you are finished fetching from a cursor or cursor variable, and that the resources held by the cursor can be reused.

Syntax

close_statement



Keyword and Parameter Description

cursor_name, cursor_variable_name, host_cursor_variable_name

When you close the cursor, you can specify an explicit cursor or a PL/SQL cursor variable, previously declared within the current scope and currently open.

You can also specify a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind 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. You must close a cursor before opening it again, otherwise PL/SQL raises the predefined exception `CURSOR_ALREADY_OPEN`. You do not need to close a cursor variable before opening it again.

If you try to close an already-closed or never-opened cursor or cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Example

```
DECLARE
   CURSOR emp_cv IS SELECT * FROM employees WHERE first_name = 'John';
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_cv;
   LOOP
       FETCH emp_cv INTO emp_rec;
       EXIT WHEN emp_cv%NOTFOUND;
   END LOOP;
   CLOSE emp_cv; /* Close cursor variable after last row is processed. */
END;
/
```

Related Topics

FETCH Statement, OPEN Statement, OPEN-FOR Statement, "Querying Data with PL/SQL" on page 6-9.

Collection Methods

A collection method is a built-in function or procedure that operates on collections and is called using dot notation. You can use the methods `EXISTS`, `COUNT`, `LIMIT`, `FIRST`, `LAST`, `PRIOR`, `NEXT`, `EXTEND`, `TRIM`, and `DELETE` to manage collections whose size is unknown or varies.

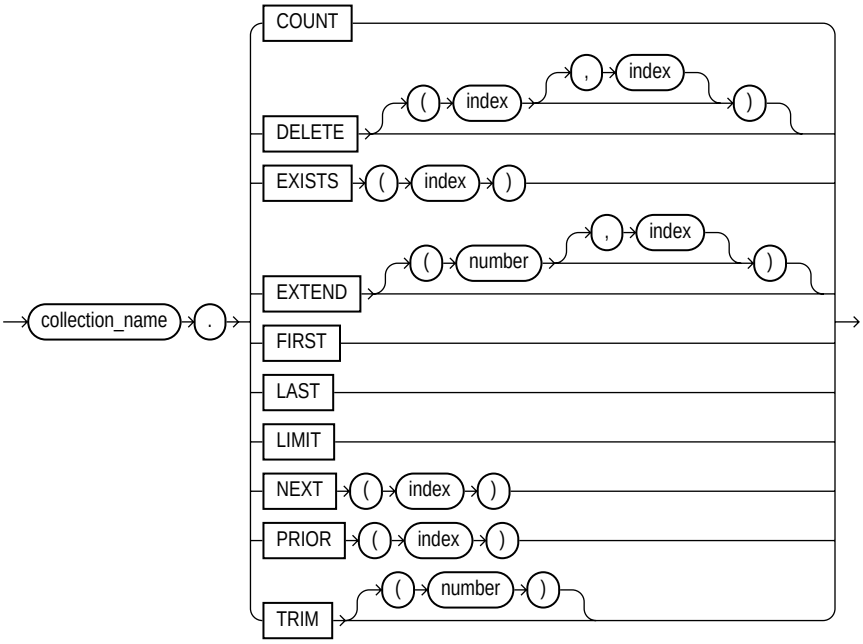
`EXISTS`, `COUNT`, `LIMIT`, `FIRST`, `LAST`, `PRIOR`, and `NEXT` are functions that check the properties of a collection or individual collection elements. `EXTEND`, `TRIM`, and `DELETE` are procedures that modify a collection.

`EXISTS`, `PRIOR`, `NEXT`, `TRIM`, `EXTEND`, and `DELETE` take integer parameters. `EXISTS`, `PRIOR`, `NEXT`, and `DELETE` can also take `VARCHAR2` parameters for associative arrays with string keys. `EXTEND` and `TRIM` cannot be used with index-by tables.

For more information, see "Using Collection Methods" on page 5-23.

Syntax

collection_method_call



Keyword and Parameter Description

collection_name

An associative array, nested table, or varray previously declared within the current scope.

COUNT

Returns the number of elements that a collection currently contains, which is useful because the current size of a collection is not always known. You can use `COUNT` wherever an integer expression is allowed.

For varrays, COUNT always equals LAST. For nested tables, normally, COUNT equals LAST. But, if you delete elements from the middle of a nested table, COUNT is smaller than LAST.

DELETE

This procedure has three forms. DELETE removes all elements from a collection. DELETE(n) removes the *n*th element from an associative array or nested table. If n is null, DELETE(n) does nothing. DELETE(m,n) removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing.

EXISTS

EXISTS(n) returns TRUE if the *n*th element in a collection exists. Otherwise, EXISTS(n) returns FALSE. Mainly, you use EXISTS with DELETE to maintain sparse nested tables. You can also use EXISTS to avoid raising an exception when you reference a nonexistent element. When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

EXTEND

This procedure has three forms. EXTEND appends one null element to a collection. EXTEND(n) appends n null elements to a collection. EXTEND(n,i) appends n copies of the *i*th element to a collection.

EXTEND operates on the internal size of a collection. If EXTEND encounters deleted elements, it includes them in its tally.

You cannot use EXTEND with associative arrays.

FIRST, LAST

FIRST and LAST return the first and last (smallest and largest) subscript values in a collection. The subscript values are usually integers, but can also be strings for associative arrays. If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same subscript value.

For varrays, FIRST always returns 1 and LAST always equals COUNT. For nested tables, normally, LAST equals COUNT. But, if you delete elements from the middle of a nested table, LAST is larger than COUNT.

index

An expression that must return (or convert implicitly to) an integer in most cases, or a string for an associative array declared with string keys.

LIMIT

For nested tables, which have no maximum size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain (which you must specify in its type definition).

NEXT, PRIOR

PRIOR(n) returns the subscript that precedes index n in a collection. NEXT(n) returns the subscript 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. If `TRIM` encounters deleted elements, it includes them in its tally.

Usage Notes

You cannot use collection methods in a SQL statement. If you try, you get a compilation error.

Only `EXISTS` can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises `COLLECTION_IS_NULL`.

If the collection elements have sequential subscripts, you can use `collection.FIRST .. collection.LAST` in a `FOR` loop to iterate through all the elements.

You can use `PRIOR` or `NEXT` to traverse collections indexed by any series of subscripts. For example, you can use `PRIOR` or `NEXT` to traverse a nested table from which some elements have been deleted, or an associative array where the subscripts are string values.

`EXTEND` operates on the internal size of a collection, which includes deleted elements. You cannot use `EXTEND` to initialize an atomically null collection. Also, if you impose the `NOT NULL` constraint on a `TABLE` or `VARRAY` type, you cannot apply the first two forms of `EXTEND` to collections of that type.

If an element to be deleted does not exist, `DELETE` simply skips it; no exception is raised. Varrays are dense, so you cannot delete their individual elements.

Because PL/SQL keeps placeholders for deleted elements, you can replace a deleted element by assigning it a new value. However, PL/SQL does not keep placeholders for trimmed elements.

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

In general, do not depend on the interaction between `TRIM` and `DELETE`. It is better to treat nested tables like fixed-size arrays and use only `DELETE`, or to treat them like stacks and use only `TRIM` and `EXTEND`.

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. You can apply methods `FIRST`, `LAST`, `COUNT`, and so on to such parameters. For varray parameters, the value of `LIMIT` is always derived from the parameter type definition, regardless of the parameter mode.

Examples

The following example shows all the collection methods in action:

```
DECLARE
   TYPE color_typ IS TABLE OF VARCHAR2(32);
   colors color_typ;
   i INTEGER;
BEGIN
   colors := color_typ('red','orange','yellow','green','blue','indigo','violet');
```

```
-- Using NEXT is the most reliable way to loop through all elements.
   i := colors.FIRST;  -- get subscript of first element
   WHILE i IS NOT NULL LOOP
      colors(i) := INITCAP(colors(i));
      dbms_output.put_line('COLORS(' || i || ') = ' || colors(i));
      i := colors.NEXT(i);  -- get subscript of next element
   END LOOP;

   dbms_output.put_line('Deleting yellow...');

   colors.DELETE(3); -- Remove item 3. Now the subscripts are 1,2,4,5,6,7.

-- Loop goes from 1 to 7, even though item 3 has been deleted.
   FOR i IN colors.FIRST..colors.LAST
   LOOP
      IF colors.EXISTS(i) THEN
         dbms_output.put_line('COLORS(' || i || ') still exists.');
      ELSE
         dbms_output.put_line('COLORS(' || i || ') no longer exists.');
      END IF;
   END LOOP;

   dbms_output.put_line('Deleting blue, indigo, violet...');
   colors.DELETE(5,7); -- Delete items 5 through 7.

-- Loop now goes from 1 to 4, because 4 is the highest ("last") subscript.
   FOR i IN colors.FIRST..colors.LAST LOOP
      IF colors.EXISTS(i) THEN
         dbms_output.put_line('COLORS(' || i || ') still exists.');
      ELSE
         dbms_output.put_line('COLORS(' || i || ') no longer exists.');
      END IF;
   END LOOP;
END;
/
```

The following example uses the `LIMIT` method to check whether some elements can be added to a varray:

```
DECLARE
   TYPE chores_typ is VARRAY(4) OF VARCHAR2(32);
   chores chores_typ;
BEGIN
   chores := chores_typ('Mow lawn','Wash dishes','Buy groceries');
   IF (chores.COUNT + 5) <= chores.LIMIT THEN
      -- Add 5 more to-do items
      dbms_output.put_line('Adding 5 more items to CHORES.');
      chores.EXTEND(5);
   ELSE
      dbms_output.put_line('Can''t extend CHORES, it can hold a maximum of ' ||
         chores.LIMIT || ' items.');
   END IF;
END;
/
```

Related Topics

Collections, Functions, Procedures

Collections

A collection is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers three kinds of collections: associative arrays, nested tables, and varrays (short for variable-size arrays). Nested tables extend the functionality of associative arrays (formerly called "PL/SQL tables" or "index-by tables").

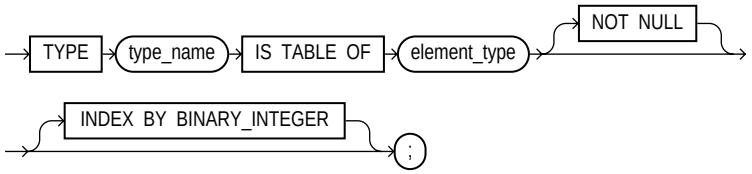
Collections work like the arrays found in most third-generation programming languages. Collections can have only one dimension. Most collections are indexed by integers, although associative arrays can also be indexed by strings. To model multi-dimensional arrays, you can declare collections whose items are other collections.

Nested tables and varrays can store instances of an object type and, conversely, can be attributes of an object type. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

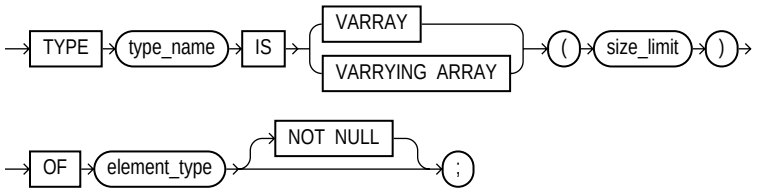
For more information, see "Defining Collection Types" on page 5-6.

Syntax

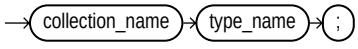
table_type_definition



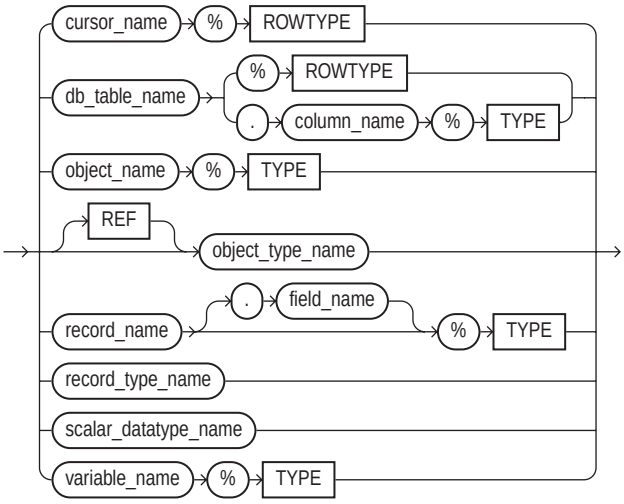
varray_type_definition



collection_declaration



element_type



Keyword and Parameter Description

element_type

Any PL/SQL datatype except `BINARY_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `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 type_name

Optional. Defines an associative array, where you specify the subscript values to use rather than the system defining them in sequence.

`type_name` can be `BINARY_INTEGER`, `PLS_INTEGER`, or a string type such as `VARCHAR2`.

size_limit

A positive integer literal that specifies the maximum size of a varray, which is the maximum number of elements the varray can contain.

type_name

A user-defined collection type that was defined using the datatype specifier `TABLE` or `VARRAY`.

Usage Notes

Nested tables extend the functionality of index-by tables, so they differ in several ways. See "Choosing Between Nested Tables and Associative Arrays" on page 5-5.

Every element reference includes the collection name and one or more subscripts enclosed in parentheses; the subscripts determine which element is processed. Except for associative arrays, which can have negative subscripts, collection subscripts have a fixed lower bound of 1. Subscripts for multilevel collections are evaluated in any order; if a subscript includes an expression that modifies the value of a different subscript, the result is undefined.

You can define all three collection types in the declarative part of any PL/SQL block, subprogram, or package. But, only nested table and varray types can be CREATEd and stored in an Oracle database.

Associative arrays and nested tables can be sparse (have 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, associative arrays are sparse. That enables you, for example, to store reference data in a temporary variable using a primary key (account numbers or employee numbers for example) as the index.

Collections follow the usual scoping and instantiation rules. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session. In a block or subprogram, local collections are instantiated when you enter the block or subprogram and cease to exist when you exit.

Until you initialize it, a nested table or varray is atomically null (that is, the collection itself is null, not its elements). To initialize a nested table or varray, you use a constructor, which is a system-defined function with the same name as the collection type. This function "constructs" 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. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

When calling a function that returns a collection, you use the following syntax to reference elements in the collection:

```
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:

```
DECLARE
   TYPE JobList IS VARRAY(10) OF employees.job_id%TYPE;  -- based on column
   TYPE EmpFile IS VARRAY(150) OF employees%ROWTYPE;  -- based on database table
   CURSOR c1 IS SELECT * FROM departments;
   TYPE DeptFile IS TABLE OF c1%ROWTYPE;  -- based on cursor
BEGIN
   NULL;
END;
/
```

You can use a RECORD type to specify the element type of a collection:

```
DECLARE
   TYPE Entry IS RECORD (
      term    VARCHAR2(20),
      meaning VARCHAR2(200));
   TYPE Glossary IS VARRAY(250) OF Entry;
```

```
BEGIN
   NULL;
END;
/
```

The following example declares an associative array of records. Each element of the table stores a row from a database table.

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab EmpTabTyp;
BEGIN
   /* Retrieve employee record. */
   SELECT * INTO emp_tab(100) FROM employees WHERE employee_id = 100;
END;
/
```

When defining a VARRAY type, you must specify its maximum size. The following example defines a type that stores up to 366 dates:

```
DECLARE
   TYPE Calendar IS VARRAY(366) OF DATE;
BEGIN
   NULL;
END;
/
```

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 temp_department (
   idnum    NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
   projects ProjectList);

DROP TABLE temp_department;
DROP TYPE ProjectList;
DROP TYPE Project;
```

The identifier `projects` represents an entire varray. Each element of `projects` stores a `Project` object.

The following example declares a nested table as the parameter of a packaged procedure:

```
CREATE PACKAGE personnel AS
   TYPE Staff IS TABLE OF Employee;
   PROCEDURE award_bonuses (members IN Staff);
END personnel;
/
```

```
DROP PACKAGE personnel;
```

You can specify a collection type in the RETURN clause of a function spec:

```
DECLARE
   TYPE SalesForce IS VARRAY(20) OF employees%ROWTYPE;
   FUNCTION top_performers (n INTEGER) RETURN SalesForce IS
       BEGIN RETURN NULL; END;
BEGIN
   NULL;
END;
/
```

The following example updates the list of projects assigned to one department:

```
-- Needs to be simplified...

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;
/
```

The next example retrieves a varray in a database table and stores it in a local varray:

```
-- Needs to be simplified...

DECLARE
   my_projects ProjectList;
BEGIN
   SELECT projects INTO my_projects FROM department
      WHERE name = 'Accounting';
END;
/
```

Related Topics

Collection Methods, Object Types, Records

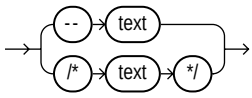
Comments

Comments let you include arbitrary text within your code to explain what the code does. You can also disable obsolete or unfinished pieces of code by turning them into comments.

PL/SQL supports two comment styles: single-line and multi-line. A double hyphen (`--`) anywhere on a line (except within a character literal) turns the rest of the line into a comment. Multi-line comments begin with a slash-asterisk (`/*`) and end with an asterisk-slash (`*/`). For more information, see "Comments" on page 2-7.

Syntax

comment



Usage Notes

Single-line comments can appear within a statement at the end of a line.

You can include single-line comments inside multi-line comments, but you cannot nest multi-line comments.

You cannot use single-line comments in a PL/SQL block that will be processed dynamically by an Oracle Precompiler program. End-of-line characters are ignored, making the single-line comments extend to the end of the block. Instead, use multi-line comments.

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 department SET location_id = my_loc WHERE department_id = my_deptno;
```

You can use multi-line comment delimiters to comment-out whole sections of code.

Examples

The following examples show various comment styles:

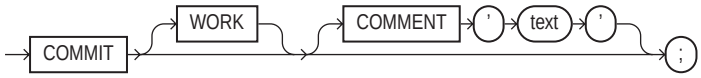
```
DECLARE
   area NUMBER; pi NUMBER; radius NUMBER;
BEGIN
   -- Compute the area of a circle
   area := pi * radius**2;  -- pi is approx. 3.14159

   /*
   Compute the area
   of a circle.
   */
   area := pi /* pi is approx. 3.14159 */ * radius**2;
END;
/
```

COMMIT Statement

The `COMMIT` statement makes permanent any changes made to the database during the current transaction. A commit also makes the changes visible to other users. For more information, see "Overview of Transaction Processing in PL/SQL" on page 6-29.

Syntax

commit_statement



Keyword and Parameter Description

COMMENT

Specifies a comment to be associated with the current transaction. Typically used with distributed transactions. The text must be a quoted literal no more than 50 characters long.

WORK

Optional, for readability only.

Usage Notes

The `COMMIT` statement releases all row and table locks, and erases any savepoints you marked since the last commit or rollback. Until your changes are committed:

- You can see the changes when you query the tables you modified, but other users cannot see the changes.
- If you change your mind or need to correct a mistake, you can use the `ROLLBACK` statement to roll back (undo) the changes.

If you commit while a `FOR UPDATE` cursor is open, a subsequent fetch on that cursor raises an exception. The cursor remains open, so you should still close it. For more information, see "Using FOR UPDATE" on page 6-33.

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 *Oracle Database Concepts*.

In SQL, the `FORCE` clause manually commits an in-doubt distributed transaction. PL/SQL does not support this clause:

```
COMMIT WORK FORCE '23.51.54';  -- not allowed
```

In embedded SQL, the `RELEASE` option frees all ocks and cursors held by a program and disconnects from the database. PL/SQL does not support this option:

```
COMMIT WORK RELEASE;  -- not allowed
```

Related Topics

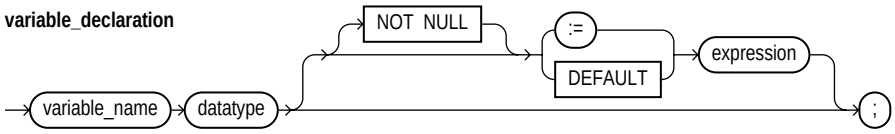
ROLLBACK Statement, SAVEPOINT Statement

Constants and Variables

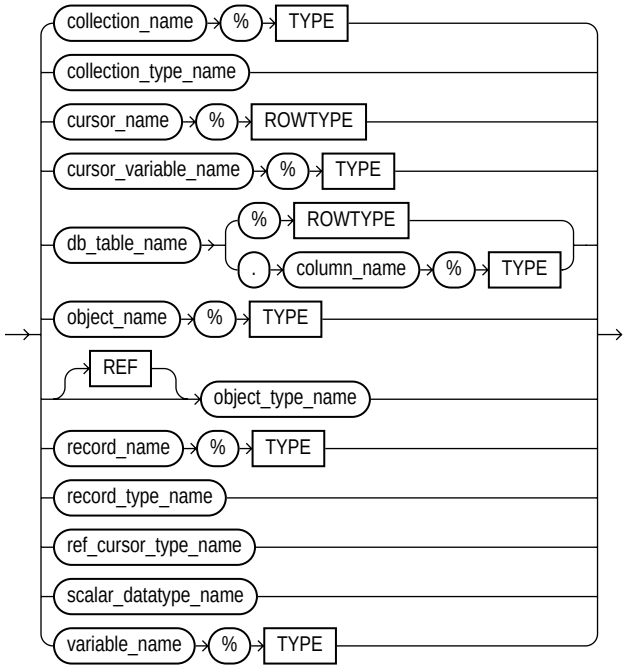
You can declare constants and variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage for a value, specify its datatype, and specify a name that you can reference. Declarations can also assign an initial value and impose the `NOT NULL` constraint. For more information, see Declarations on page 2-8.

Syntax

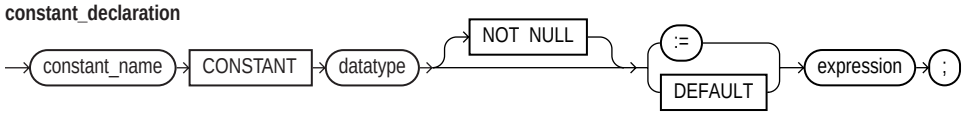
variable_declaration



datatype



constant_declaration



Keyword and Parameter Description

collection_name

A collection (associative array, nested table, or varray) previously declared within the current scope.

collection_type_name
A user-defined collection type defined using the datatype specifier `TABLE` or `VARRAY`.

CONSTANT
Denotes the declaration of a constant. You must initialize a constant in its declaration. Once initialized, the value of a constant cannot be changed.

constant_name
A program constant. For naming conventions, see "Identifiers" on page 2-3.

cursor_name
An explicit cursor previously declared within the current scope.

cursor_variable_name
A PL/SQL cursor variable previously declared within the current scope.

db_table_name
A database table or view that must be accessible when the declaration is elaborated.

db_table_name.column_name
A database table and column that must be accessible when the declaration is elaborated.

expression
A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the constant or variable. The value and the constant or variable must have compatible datatypes.

NOT NULL
A constraint that prevents the program from assigning a null value to a variable or constant. Assigning a null to a variable defined as `NOT NULL` raises the predefined exception `VALUE_ERROR`. The constraint `NOT NULL` must be followed by an initialization clause.

object_name
An instance of an object type previously declared within the current scope.

record_name
A user-defined or `%ROWTYPE` record previously declared within the current scope.

record_name.field_name
A field in a user-defined or `%ROWTYPE` record previously declared within the current scope.

record_type_name
A user-defined record type that is defined using the datatype specifier `RECORD`.

ref_cursor_type_name
A user-defined cursor variable type, defined using the datatype specifier `REF CURSOR`.

%ROWTYPE

Represents a record that can hold a row from a database table or a cursor. Fields in the record have the same names and datatypes as columns in the row.

scalar_datatype_name

A predefined scalar datatype such as `BOOLEAN`, `NUMBER`, or `VARCHAR2`. Includes any qualifiers for size, precision, or character versus byte semantics.

%TYPE

Represents the datatype of a previously declared collection, cursor variable, field, object, record, database column, or variable.

variable_name

A program variable.

Usage Notes

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

Whether public or private, constants and variables declared in a package spec are initialized only once for each session.

An initialization clause is required when declaring `NOT NULL` variables and when declaring constants. If you use `%ROWTYPE` to declare a variable, initialization is not allowed.

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

Examples

Several examples of variable and constant declarations follow:

```
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 2-8, "Overview of Predefined PL/SQL Datatypes" on page 3-1, Assignment Statement, Expressions, %ROWTYPE Attribute, %TYPE Attribute

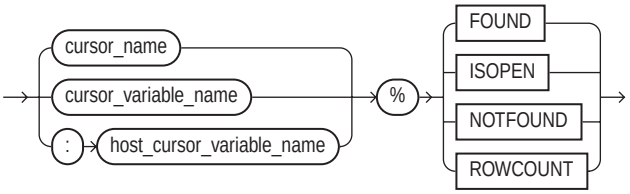
Cursor Attributes

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 Expressions" on page 6-27.

The implicit cursor `SQL` has additional attributes, `%BULK_ROWCOUNT` and `%BULK_EXCEPTIONS`. For more information, see "SQL Cursor" on page 13-131.

Syntax

cursor_attribute



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable (or parameter) previously declared within the current scope.

%FOUND Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. Before the first fetch from an open cursor, `cursor_name%FOUND` returns `NULL`. Afterward, it returns `TRUE` if the last fetch returned a row, or `FALSE` if the last fetch failed to return a row.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind 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

A cursor attribute that can be appended to the name of a cursor or cursor variable. If a cursor is open, `cursor_name%ISOPEN` returns `TRUE`; otherwise, it returns `FALSE`.

%NOTFOUND Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. Before the first fetch from an open cursor, `cursor_name%NOTFOUND` returns `NULL`. Thereafter, it returns `FALSE` if the last fetch returned a row, or `TRUE` if the last fetch failed to return a row.

%ROWCOUNT Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. When a cursor is opened, %ROWCOUNT is zeroed. Before the first fetch, cursor_name%ROWCOUNT returns 0. Thereafter, it returns the number of rows fetched so far. The number is incremented if the latest fetch returned a row.

Usage Notes

The cursor attributes apply to every cursor or cursor variable. For example, you can open multiple cursors, then use %FOUND or %NOTFOUND to tell which cursors have rows left to fetch. Likewise, you can use %ROWCOUNT to tell how many rows 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. If FETCH never executes successfully, the EXIT WHEN condition is never TRUE and the loop is never exited. 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

This PL/SQL block uses %FOUND to select an action.

```
DECLARE
   CURSOR emp_cur IS SELECT * FROM employees ORDER BY employee_id;
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_cur;
   LOOP   -- loop through the table and get each employee
      FETCH emp_cur INTO emp_rec;
      IF emp_cur%FOUND THEN
         dbms_output.put_line('Employee #' || emp_rec.employee_id ||
            ' is ' || emp_rec.last_name);
      ELSE
         dbms_output.put_line('--- Finished processing employees ---');
         EXIT;
      END IF;
   END LOOP;
   CLOSE emp_cur;
END;
/
```

Instead of using %FOUND in an IF statement, the next example uses %NOTFOUND in an EXIT WHEN statement.

```
DECLARE
   CURSOR emp_cur IS SELECT * FROM employees ORDER BY employee_id;
   emp_rec employees%ROWTYPE;
```

```
BEGIN
   OPEN emp_cur;
   LOOP   -- loop through the table and get each employee
      FETCH emp_cur INTO emp_rec;
      EXIT WHEN emp_cur%NOTFOUND;
      dbms_output.put_line('Employee #' || emp_rec.employee_id ||
         ' is ' || emp_rec.last_name);
   END LOOP;
   CLOSE emp_cur;
END;
/
```

The following example uses `%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:

```
DECLARE
   CURSOR c1 is
   SELECT last_name, employee_id, salary FROM employees
      ORDER BY salary DESC;   -- start with highest-paid employee
   my_name employees.last_name%TYPE;
   my_empno employees.employee_id%TYPE;
   my_sal   employees.salary%TYPE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_name, my_empno, my_sal;
      EXIT WHEN (c1%ROWCOUNT > 5) OR (c1%NOTFOUND);
      dbms_output.put_line('Employee ' || my_name || ' (' || my_empno || ') makes
' || my_sal);
   END LOOP;
   CLOSE c1;
END;
/
```

The following example raises an exception if many rows are 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

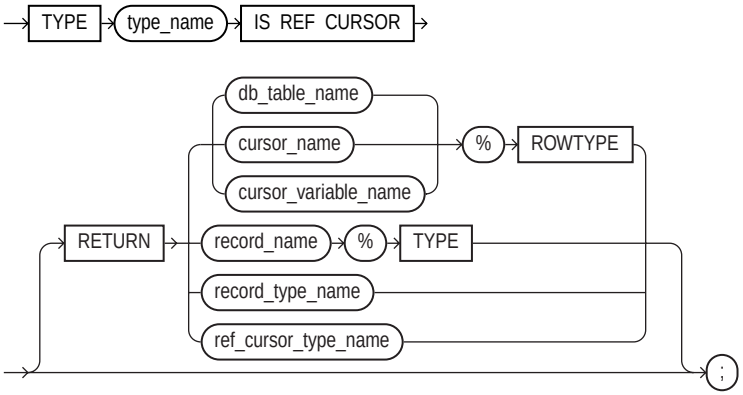
To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. You can access this area through an explicit cursor, which names the work area, or through a cursor variable, which points to the work area. To create cursor variables, you define a `REF CURSOR` type, then declare cursor variables of that type.

Cursor variables are like C or Pascal pointers, which hold the address of some item instead of the item itself. Declaring a cursor variable creates a pointer, not an item.

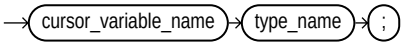
For more information, see "Using Cursor Variables (REF CURSORs)" on page 6-19.

Syntax

ref_cursor_type_definition



cursor_variable_declaration



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope.

db_table_name

A database table or view, which must be accessible when the declaration is elaborated.

record_name

A user-defined record previously declared within the current scope.

record_type_name

A user-defined record type that was defined using the datatype specifier `RECORD`.

REF CURSOR

Cursor variables all have the datatype `REF CURSOR`.

RETURN

Specifies the datatype of a cursor variable return value. You can use the `%ROWTYPE` attribute in the `RETURN` clause to provide a record type that represents a row in a database table, or a row from a cursor or strongly typed cursor variable. You can use the `%TYPE` attribute to provide the datatype of a previously declared record.

%ROWTYPE

A record type that represents a row in a database table or a row fetched from a cursor or strongly typed cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.

%TYPE

Provides the datatype of a previously declared user-defined record.

type_name

A user-defined cursor variable type that was defined as a `REF CURSOR`.

Usage Notes

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as a bind variable to PL/SQL. Application development tools that have a PL/SQL engine can use cursor variables entirely on the client side.

You can pass cursor variables back and forth between an application and the database server through remote procedure calls using a database link. If you have a PL/SQL engine on the client side, you can use the cursor variable in either location. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side.

You use cursor variables to pass query result sets between PL/SQL stored subprograms and client programs. Neither PL/SQL nor any client program owns a result set; they share a pointer to the work area where the result set is stored. For example, an OCI program, Oracle Forms application, and the database can all refer to the same work area.

`REF CURSOR` types can be *strong* or *weak*. A strong `REF CURSOR` type definition specifies a return type, but a weak definition does not. Strong `REF CURSOR` types are less error-prone because PL/SQL lets you associate a strongly typed cursor variable only with type-compatible queries. Weak `REF CURSOR` types are more flexible because you can associate a weakly typed cursor variable with any query.

Once you define a `REF CURSOR` type, you can declare cursor variables of that type. You can use `%TYPE` to provide the datatype of a record variable. Also, in the `RETURN` clause of a `REF CURSOR` type definition, you can use `%ROWTYPE` to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable.

Currently, cursor variables are subject to several restrictions. See "Restrictions on Cursor Variables" on page 6-27.

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:

```
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:

```
/* 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:

```
CREATE PACKAGE emp_data AS
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%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 employees WHERE commission_pct IS NOT NULL;
      ELSIF choice = 2 THEN
         OPEN emp_cv FOR SELECT * FROM employees WHERE salary > 2500;
      ELSIF choice = 3 THEN
         OPEN emp_cv FOR SELECT * FROM employees WHERE department_id = 20;
      END IF;
   END open_emp_cv;
END emp_data;
/
DROP PACKAGE emp_data;
```

You can also use a standalone procedure to open the cursor variable. Define the REF CURSOR type in a package, as above, then reference that type in the standalone procedure.

Related Topics

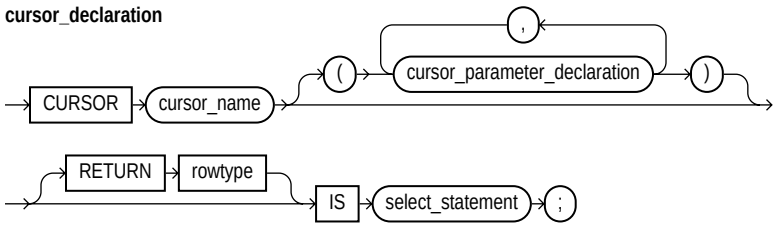
CLOSE Statement, Cursor Attributes, Cursors, FETCH Statement, OPEN-FOR Statement

Cursors

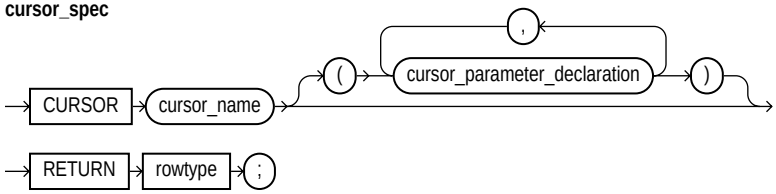
To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. A cursor lets you name the work area, access the information, and process the rows individually. For more information, see "Querying Data with PL/SQL" on page 6-9.

Syntax

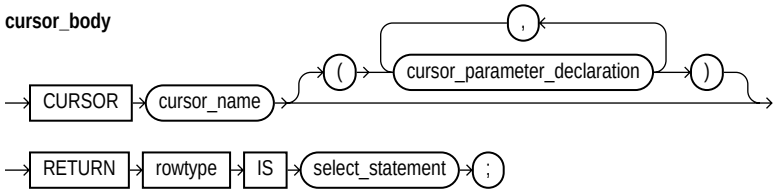
cursor_declaration



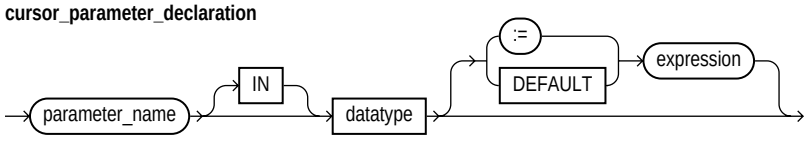
cursor_spec



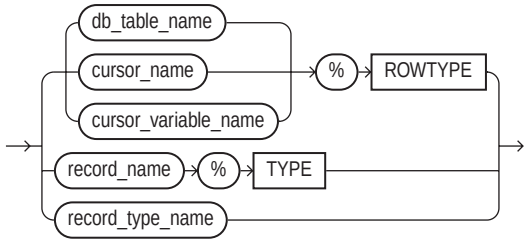
cursor_body



cursor_parameter_declaration



rowtype



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

datatype

A type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 13-28.

db_table_name

A database table or view that must be accessible when the declaration is elaborated.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

parameter_name

A variable declared as the formal parameter of a cursor. A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be `IN` parameters. The query can also reference other PL/SQL variables within its scope.

record_name

A user-defined record previously declared within the current scope.

record_type_name

A user-defined record type that was defined using the datatype specifier `RECORD`.

RETURN

Specifies the datatype of a cursor return value. You can use the `%ROWTYPE` attribute in the `RETURN` clause to provide a record type that represents a row in a database table or a row returned by a previously declared cursor. Also, you can use the `%TYPE` attribute to provide the datatype of a previously declared record.

A cursor body must have a `SELECT` statement and the same `RETURN` clause as its corresponding cursor spec. Also, the number, order, and datatypes of select items in the `SELECT` clause must match the `RETURN` clause.

%ROWTYPE

A record type that represents a row in a database table or a row fetched from a previously declared cursor or cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.

select_statement

A query that returns a result set of rows. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 13-123. If the cursor declaration declares parameters, each parameter must be used in the query.

%TYPE

Provides the datatype of a previously declared user-defined record.

Usage Notes

You must declare a cursor before referencing it in an `OPEN`, `FETCH`, or `CLOSE` statement. You must declare a variable before referencing it in a cursor declaration. The word SQL is reserved by PL/SQL as the default name for implicit cursors, and cannot be used in a cursor declaration.

You cannot assign values to a cursor name or use it in an expression. However, cursors and variables follow the same scoping rules. For more information, see "Scope and Visibility of PL/SQL Identifiers" on page 2-14.

You retrieve data from a cursor by opening it, then fetching from it. Because the `FETCH` statement specifies the target variables, using an `INTO` clause in the `SELECT` statement of a `cursor_declaration` is redundant and invalid.

The scope of cursor parameters is local to the cursor, meaning that they can be referenced only within the query used in the cursor declaration. The values of cursor parameters are used by the associated query when the cursor is opened. The query can also reference other PL/SQL variables within its scope.

The datatype of a cursor parameter must be specified without constraints, that is, without precision and scale for numbers, and without length for strings.

Examples

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;
```

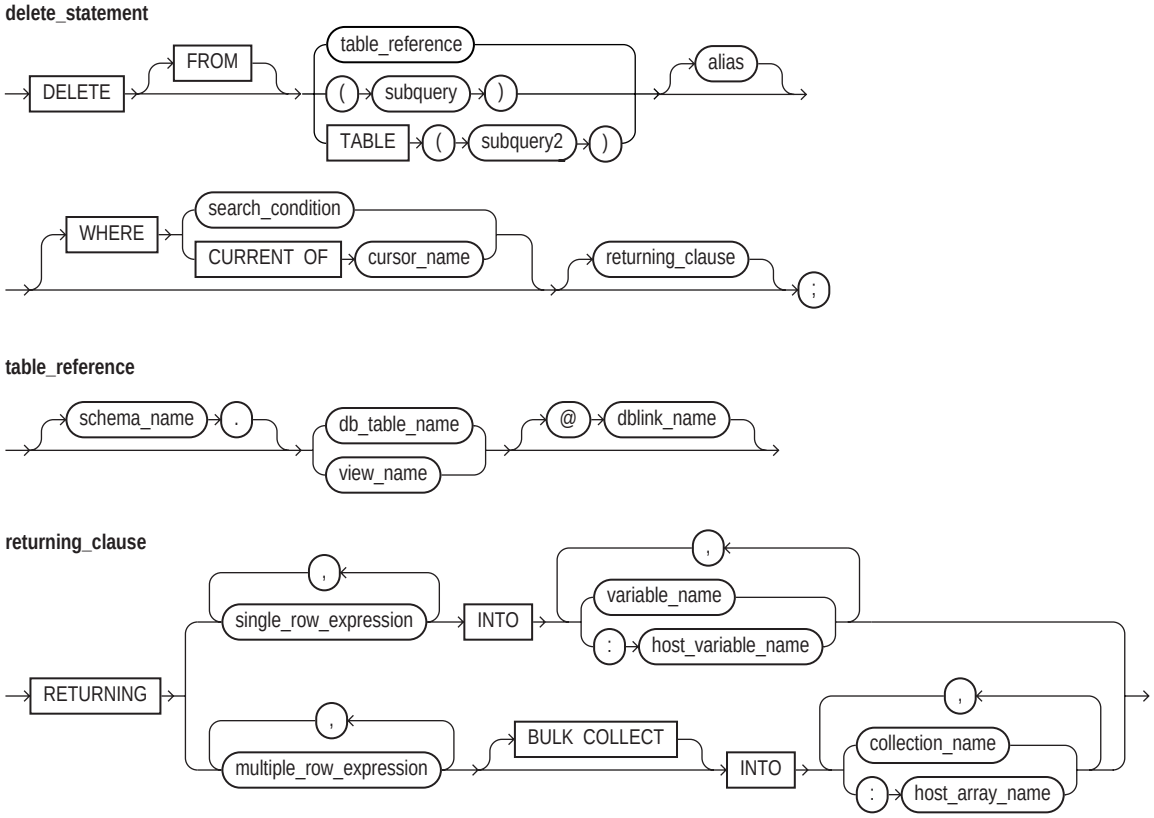
Related Topics

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 **Oracle Database SQL Reference**.

Syntax



Keyword and Parameter Description

alias

Another (usually short) name for the referenced table or view. Typically referred to later in the WHERE clause.

BULK COLLECT

Returns columns from the deleted rows into PL/SQL collections, as specified by the RETURNING INTO list. The corresponding columns must store scalar (not composite) values. For more information, see "Reducing Loop Overhead for DML Statements and Queries (FORALL, BULK COLLECT)" on page 11-7.

returning_clause

Returns values from the deleted rows, eliminating the need to SELECT the rows first. You can retrieve the column values into individual variables or into collections. You cannot use the RETURNING clause for remote or parallel deletes. If the statement does not affect any rows, the values of the variables specified in the RETURNING clause are undefined.

subquery

A SELECT statement that provides a set of rows for processing. Its syntax is like the select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 13-123.

table_reference

A table or view, which must be accessible when you execute the DELETE statement, and for which you must have DELETE privileges.

TABLE (subquery2)

The operand of TABLE is a SELECT statement that returns a single column value, which must be a nested table. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

WHERE CURRENT OF cursor_name

Refers to the latest row processed by the FETCH statement associated with the cursor identified by cursor_name. The cursor must be FOR UPDATE and must be open and positioned on a row. If the cursor is not open, the CURRENT OF clause causes an error.

If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception NO_DATA_FOUND.

WHERE search_condition

Conditionally chooses rows to be deleted from the referenced table or view. Only rows that meet the search condition are deleted. If you omit the WHERE clause, all rows in the table or view are deleted.

Usage Notes

You can use the DELETE WHERE CURRENT OF statement after a fetch from an open cursor (this includes implicit fetches executed in a cursor FOR loop), provided the associated query is FOR UPDATE. This statement deletes the current row; that is, the one just fetched.

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND, and %ROWCOUNT let you access useful information about the execution of a DELETE statement.

Examples

The following statement deletes the rows that match a condition:

```
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:

```
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

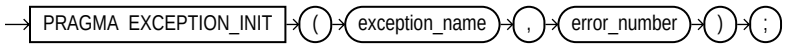
FETCH Statement, INSERT Statement, SELECT INTO Statement, UPDATE Statement

EXCEPTION_INIT Pragma

The pragma `EXCEPTION_INIT` associates an exception name with an Oracle error number. You can intercept any ORA- error and write a specific handler for it instead of using the `OTHERS` handler. For more information, see "Associating a PL/SQL Exception with a Number: Pragma EXCEPTION_INIT" on page 10-7.

Syntax

exception_init_pragma



Keyword and Parameter Description

error_number

Any valid Oracle error number. These are the same error numbers (always negative) returned by the function `SQLCODE`.

exception_name

A user-defined exception declared within the current scope.

PRAGMA

Signifies that the statement is a compiler directive.

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

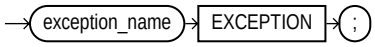
AUTONOMOUS_TRANSACTION Pragma, Exceptions, SQLCODE Function

Exceptions

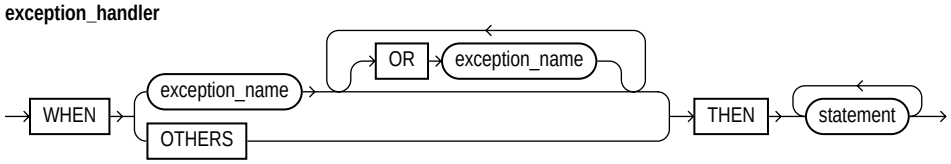
An exception is a runtime error or warning condition, which can be predefined or user-defined. Predefined exceptions are raised implicitly (automatically) by the runtime system. User-defined exceptions must be raised explicitly by `RAISE` statements. To handle raised exceptions, you write separate routines called exception handlers. For more information, see Chapter 10.

Syntax

exception_declaration



exception_handler



Keyword and Parameter Description

exception_name

A predefined exception such as `ZERO_DIVIDE`, or a user-defined exception previously declared within the current scope.

OTHERS

Stands for all the exceptions not explicitly named in the exception-handling part of the block. The use of `OTHERS` is optional and is allowed only as the last exception handler. You cannot include `OTHERS` in a list of exceptions following the keyword `WHEN`.

statement

An executable statement. For the syntax of `statement`, see "Blocks" on page 13-8.

WHEN

Introduces an exception handler. You can have multiple exceptions execute the same sequence of statements by following the keyword `WHEN` with a list of the exceptions, separating them by the keyword `OR`. If any exception in the list is raised, the associated statements are executed.

Usage Notes

An exception declaration can appear only in the declarative part of a block, subprogram, or package. The scope rules for exceptions and variables are the same. But, unlike variables, exceptions cannot be passed as parameters to subprograms.

Some exceptions are predefined by PL/SQL. For a list of these exceptions, see "Summary of Predefined PL/SQL Exceptions" on page 10-4. 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:

```
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

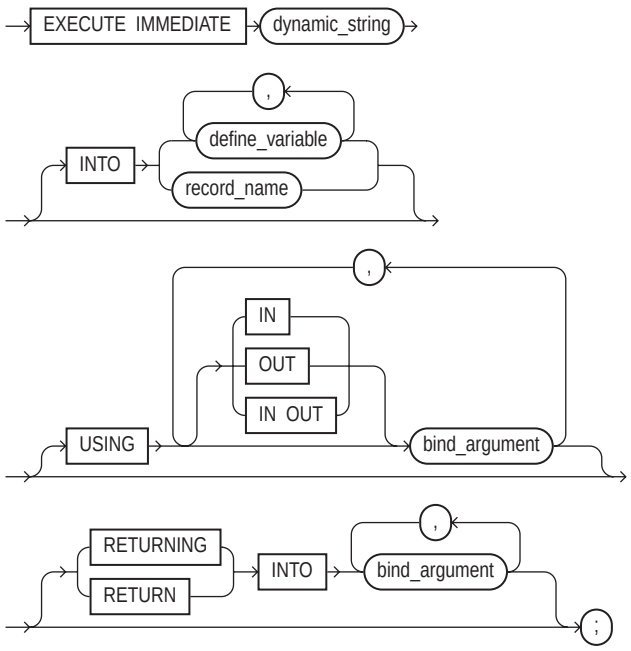
Blocks, EXCEPTION_INIT Pragma, RAISE Statement

EXECUTE IMMEDIATE Statement

The `EXECUTE IMMEDIATE` statement executes a dynamic SQL statement or anonymous PL/SQL block. You can use it to issue SQL statements that cannot be represented directly in PL/SQL, or to build up statements where you do not know all the table names, WHERE clauses, and so on in advance. For more information, see Chapter 7.

Syntax

execute_immediate_statement



Keyword and Parameter Description

bind_argument

An expression whose value is passed to the dynamic SQL statement, or a variable that stores a value returned by the dynamic SQL statement.

define_variable_name

A variable that stores a selected column value.

dynamic_string

A string literal, variable, or expression that represents a single SQL statement or a PL/SQL block. It must be of type `CHAR` or `VARCHAR2`, not `NCHAR` or `NVARCHAR2`.

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

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 ...

Specifies a list of input and/or output bind arguments. The parameter mode defaults to IN.

Usage Notes

Except for multi-row queries, the dynamic string can contain any SQL statement (*without* the final semicolon) or any PL/SQL block (with the final semicolon). The string can also contain placeholders for bind arguments. 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.

At run time, bind arguments replace corresponding placeholders in the dynamic string. 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 to Dynamic SQL" on page 7-10.

Dynamic SQL supports all the SQL datatypes. For example, define variables and bind arguments can be collections, LOBs, instances of an object type, and refs. Dynamic SQL does not support PL/SQL-specific types. 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. You still incur some overhead, because EXECUTE IMMEDIATE re-prepares the dynamic string before every execution.

The string argument to the EXECUTE IMMEDIATE command cannot be one of the national character types, such as NCHAR or NVARCHAR2.

Examples

The following PL/SQL block contains several examples of dynamic SQL:

```
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

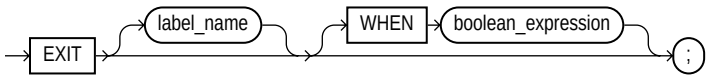
OPEN-FOR-USING Statement

EXIT Statement

The EXIT statement breaks out of 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 "Controlling Loop Iterations: LOOP and EXIT Statements" on page 4-6.

Syntax

exit_statement



Keyword and Parameter Description

boolean_expression

An expression that returns the Boolean value TRUE, FALSE, or NULL. It is evaluated with each iteration of the loop. If the expression returns TRUE, the current loop (or the loop labeled by label_name) is exited immediately. For the syntax of boolean_expression, see "Expressions" on page 13-52.

EXIT

An unconditional EXIT statement (that is, one without a WHEN clause) exits the current loop immediately. Execution resumes with the statement following the loop.

label_name

Identifies the loop exit from: either the current loop, or any enclosing labeled loop.

Usage Notes

The EXIT statement can be used only inside a loop. PL/SQL lets you code an infinite loop. For example, the following loop will never terminate normally:

```
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 not allowed because you cannot exit from a block directly; you can exit only from a loop:

```
DECLARE
   amount  NUMBER;
   maximum NUMBER;
BEGIN
   ...
   BEGIN
      ...
      IF amount >= maximum THEN
```

```
         EXIT;  -- not allowed; use RETURN instead
      END IF;
   END;
```

The following loop normally executes ten times, but it will exit prematurely if there are less than ten rows to fetch:

```
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:

```
<<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

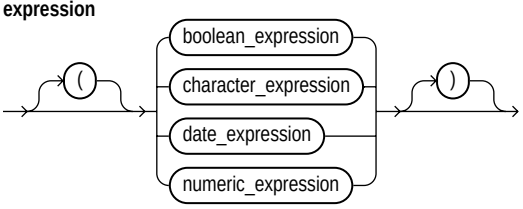
Expressions

An expression is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression is a single variable.

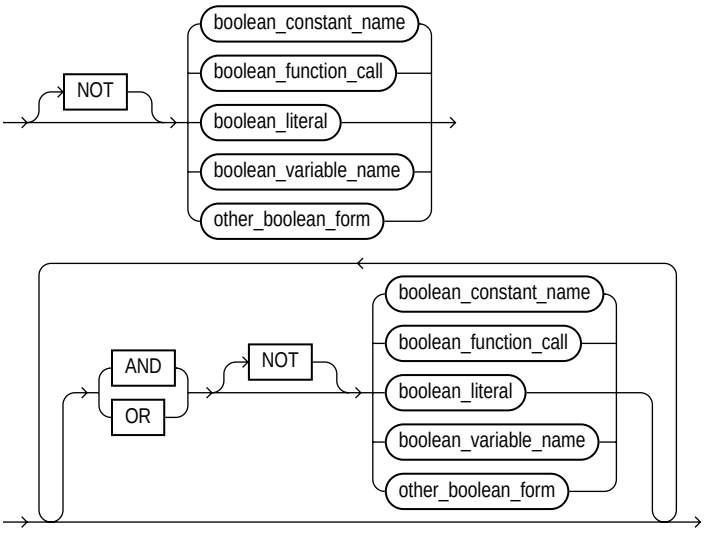
The PL/SQL compiler determines the datatype of an expression from the types of the variables, constants, literals, and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results. For more information, see "PL/SQL Expressions and Comparisons" on page 2-17.

Syntax

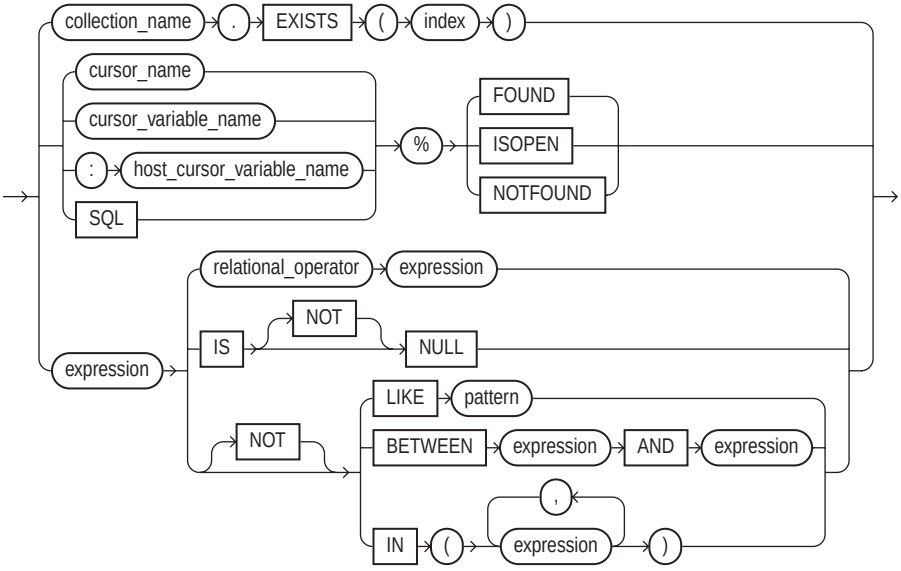
expression



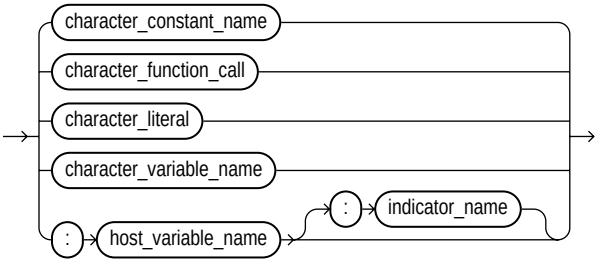
boolean_expression

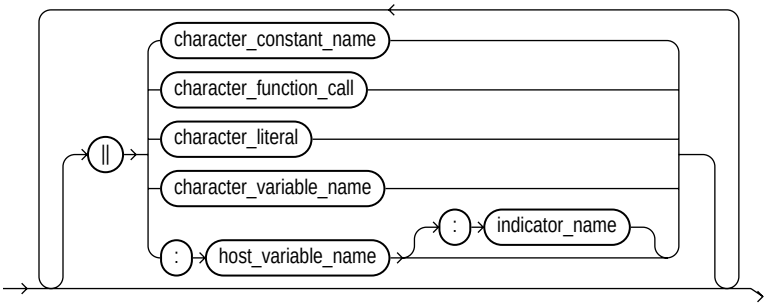


other_boolean_form

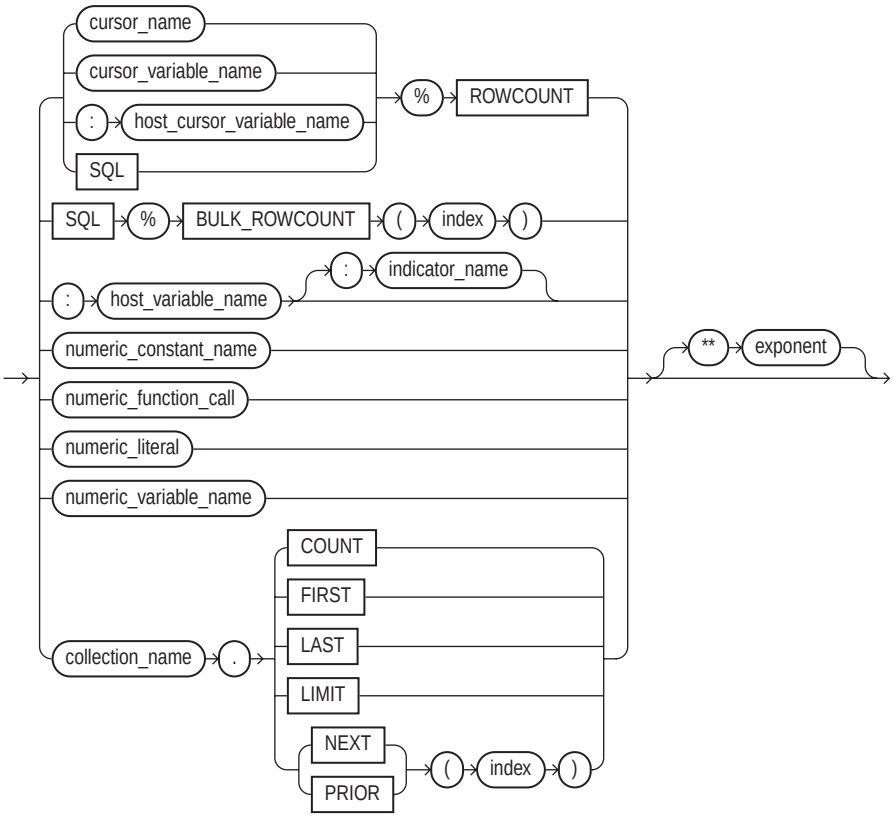


character_expression

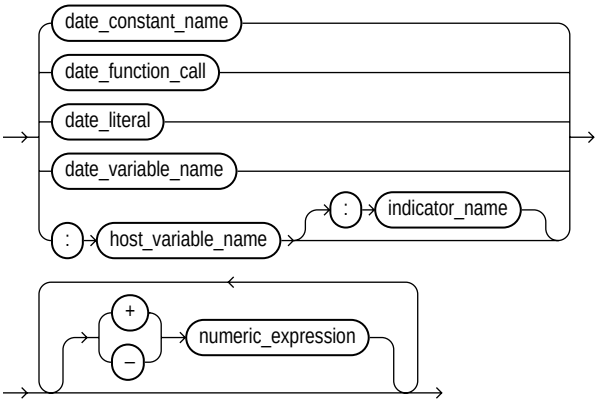




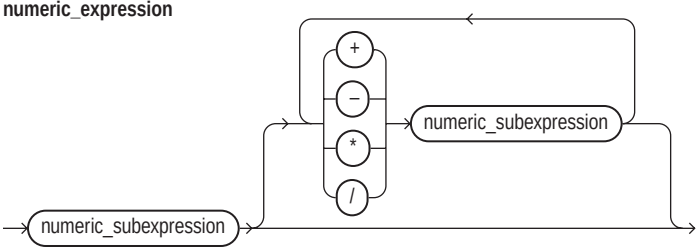
numeric_subexpression



date_expression



numeric_expression



Keyword and Parameter Description

BETWEEN

This comparison operator tests whether a value lies in a specified range. It means "greater than or equal to *low value* and less than or equal to *high value*."

boolean_constant_name

A constant of type BOOLEAN, which must be initialized to the value TRUE, FALSE, or NULL. Arithmetic operations on Boolean constants are not allowed.

boolean_expression

An expression that returns the Boolean value TRUE, FALSE, or NULL.

boolean_function_call

Any function call that returns a Boolean value.

boolean_literal

The predefined values TRUE, FALSE, or NULL (which stands for a missing, unknown, or inapplicable value). You cannot insert the value TRUE or FALSE into a database column.

boolean_variable_name

A variable of type BOOLEAN. Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable. You cannot select or fetch column values into a BOOLEAN variable. Also, arithmetic operations on BOOLEAN variables are not allowed.

%BULK_ROWCOUNT

Designed for use with the FORALL statement, this is a composite attribute of the implicit cursor SQL. For more information, see "SQL Cursor" on page 13-131.

character_constant_name

A previously declared constant that stores a character value. It must be initialized to a character value or a value implicitly convertible to a character value.

character_expression

An expression that returns a character or character string.

character_function_call

A function call that returns a character value or a value implicitly convertible to a character value.

character_literal

A literal that represents a character value or a value implicitly convertible to a character value.

character_variable_name

A previously declared variable that stores a character value.

collection_name

A collection (nested table, index-by table, or varray) previously declared within the current scope.

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope.

date_constant_name

A previously declared constant that stores a date value. It must be initialized to a date value or a value implicitly convertible to a date value.

date_expression

An expression that returns a date/time value.

date_function_call

A function call that returns a date value or a value implicitly convertible to a date value.

date_literal

A literal representing a date value or a value implicitly convertible to a date value.

date_variable_name

A previously declared variable that stores a date value.

EXISTS, COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR

Collection methods. When appended to the name of a collection, these methods return useful information. For example, `EXISTS(n)` returns `TRUE` if the *n*th element of a collection exists. Otherwise, `EXISTS(n)` returns `FALSE`. For more information, see "Collection Methods" on page 13-17.

exponent

An expression that must return a numeric value.

%FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT

Cursor attributes. When appended to the name of a cursor or cursor variable, these attributes return useful information about the execution of a multi-row query. You can also append them to the implicit cursor `SQL`.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Host cursor variables must be prefixed with a colon.

host_variable_name

A variable declared in a PL/SQL host environment and passed to PL/SQL as a bind 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

Comparison operator that tests set membership. It means "equal to any member of." The set can contain nulls, but they are ignored. Also, expressions of the form

```
value NOT IN set
```

return FALSE if the set contains a null.

index

A numeric expression that must return a value of type BINARY_INTEGER or a value implicitly convertible to that datatype.

indicator_name

An indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable "indicates" the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables can detect nulls or truncated values in output host variables.

IS NULL

Comparison operator that returns the Boolean value TRUE if its operand is null, or FALSE if its operand is not null.

LIKE

Comparison operator that compares a character value to a pattern. Case is significant. LIKE returns the Boolean value TRUE if the character patterns match, or FALSE if they do not match.

NOT, AND, OR

Logical operators, which follow the tri-state logic of Table 2–2 on page 2-18. 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 2-18.

NULL

Keyword that represents a null. It stands for a missing, unknown, or inapplicable value. When NULL is used in a numeric or date expression, the result is a null.

numeric_constant_name

A previously declared constant that stores a numeric value. It must be initialized to a numeric value or a value implicitly convertible to a numeric value.

numeric_expression

An expression that returns an integer or real value.

numeric_function_call

A function call that returns a numeric value or a value implicitly convertible to a numeric value.

numeric_literal

A literal that represents a number or a value implicitly convertible to a number.

numeric_variable_name

A previously declared variable that stores a numeric value.

pattern

A character string compared by the `LIKE` operator to a specified string value. It can include two special-purpose characters called wildcards. An underscore (_) matches exactly one character; a percent sign (%) matches zero or more characters. The pattern can be followed by `ESCAPE '*character_literal*'`, which turns off wildcard expansion wherever the escape character appears in the string followed by a percent sign or underscore.

relational_operator

Operator that compares expressions. For the meaning of each operator, see "Comparison Operators" on page 2-20.

SQL

A cursor opened implicitly by Oracle to process a SQL data manipulation statement. The implicit cursor `SQL` always refers to the most recently executed SQL statement.

+, -, /, *, **

Symbols for the addition, subtraction, division, multiplication, and exponentiation operators.

||

The concatenation operator. As the following example shows, the result of concatenating *string1* with *string2* is a character string that contains *string1* followed by *string2*:

```
'Good' || ' morning!' = 'Good morning!'
```

The next example shows that nulls have no effect on the result of a concatenation:

```
'suit' || NULL || 'case' = 'suitcase'
```

A null string (`''`), which is zero characters in length, is treated like a null.

Usage Notes

In a Boolean expression, you can only compare values that have compatible datatypes. For more information, see "Converting PL/SQL Datatypes" on page 3-18.

In conditional control statements, if a Boolean expression returns `TRUE`, its associated sequence of statements is executed. But, if the expression returns `FALSE` or `NULL`, its associated sequence of statements is *not* executed.

The relational operators can be applied to operands of type `BOOLEAN`. By definition, `TRUE` is greater than `FALSE`. Comparisons involving nulls always return a null. The value of a Boolean expression can be assigned only to Boolean variables, not to host variables or database columns. Also, datatype conversion to or from type `BOOLEAN` is not supported.

You can use the addition and subtraction operators to increment or decrement a date value, as the following examples show:

```
hire_date := '10-MAY-95';
hire_date := hire_date + 1;  -- makes hire_date '11-MAY-95'
hire_date := hire_date - 5;  -- makes hire_date '06-MAY-95'
```

When PL/SQL evaluates a boolean expression, NOT has the highest precedence, AND has the next-highest precedence, and OR has the lowest precedence. However, you can use parentheses to override the default operator precedence.

Within an expression, operations occur in 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:

```
(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
```

Related Topics

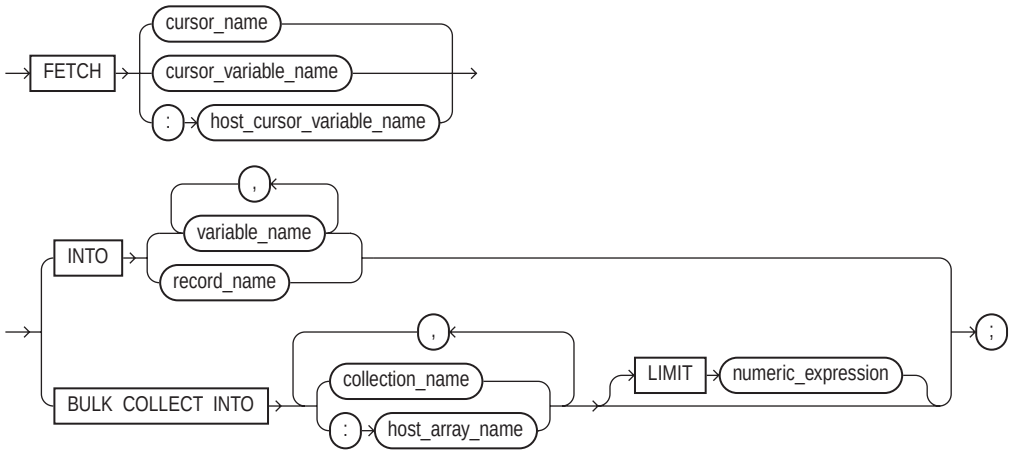
Assignment Statement, Constants and Variables, EXIT Statement, IF Statement, LOOP Statements

FETCH Statement

The `FETCH` statement retrieves rows of data from the result set of a multi-row query. You can fetch rows one at a time, several at a time, or all at once. The data is stored in variables or fields that correspond to the columns selected by the query. For more information, see "Querying Data with PL/SQL" on page 6-9.

Syntax

fetch_statement



Keyword and Parameter Description

BULK COLLECT

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

A declared collection into which column values are bulk fetched. For each query `select_item`, there must be a corresponding, type-compatible collection in the list.

cursor_name

An explicit cursor declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable (or parameter) declared within the current scope.

host_array_name

An array (declared in a PL/SQL host environment and passed to PL/SQL as a bind 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

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 bulk fetch several rows at a time, rather than the entire result set.

record_name

A user-defined or `%ROWTYPE` record into which rows of values are fetched. For each column value returned by the query associated with the cursor or cursor variable, there must be a corresponding, type-compatible field in the record.

variable_name

A variable into which a column value is fetched. For each column value returned by the query associated with the cursor or cursor variable, there must be a corresponding, type-compatible variable in the list.

Usage Notes

You must use either a cursor `FOR` loop or the `FETCH` statement to process a multi-row query.

Any variables in the `WHERE` clause of the query are evaluated only when the cursor or cursor variable is opened. To change the result set or the values of variables in the query, you must reopen the cursor or cursor variable with the variables set to their new values.

To reopen a cursor, you must close it first. However, you need not close a cursor variable before reopening it.

You can use different `INTO` lists on separate fetches with the same cursor or cursor variable. Each fetch retrieves another row and assigns values to the target variables.

If you `FETCH` past the last row in the result set, the values of the target fields or variables are indeterminate and the `%NOTFOUND` attribute returns `TRUE`.

PL/SQL makes sure the return type of a cursor variable is compatible with the `INTO` clause of the `FETCH` statement. For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible field or variable in the `INTO` clause. Also, the number of fields or variables must equal the number of column values.

When you declare a cursor variable as the formal parameter of a subprogram that fetches from the cursor variable, you must specify the `IN` or `IN OUT` mode. However, if the subprogram also opens the cursor variable, you must specify the `IN OUT` mode.

Because a sequence of `FETCH` statements always runs out of data to retrieve, no exception is raised when a FETCH returns no data. To detect this condition, you must use the cursor attribute `%FOUND` or `%NOTFOUND`.

PL/SQL raises the predefined exception `INVALID_CURSOR` if you try to fetch from a closed or never-opened cursor or cursor variable.

Restrictions on BULK COLLECT

[Moved from Collections and Tuning chapters -- might have to move it once more! -- John]

The following restrictions apply to the BULK COLLECT clause:

- You cannot bulk collect into an associative array that has a string type for the key.
- 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 target variables listed in a BULK COLLECT INTO clause must be collections.
- Composite targets (such as objects) cannot be used in the RETURNING INTO clause. Otherwise, you get the error *unsupported feature with RETURNING clause*.
- When implicit datatype conversions are needed, multiple composite targets cannot be used in the BULK COLLECT INTO clause.
- When an implicit datatype conversion is needed, a collection of a composite target (such as a collection of objects) cannot be used in the BULK COLLECT INTO clause.

Examples

The following example shows that any variables in the query associated with a cursor are evaluated only when the cursor is opened:

```
DECLARE
   my_sal NUMBER(7,2);
   n      INTEGER(2) := 2;
   CURSOR emp_cur IS SELECT  n*sal FROM emp;
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;
```

The following example fetches 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 fetch entire columns from the result set, or the entire result set at once. The following example, retrieves columns 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;
```

The following example uses 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;
```

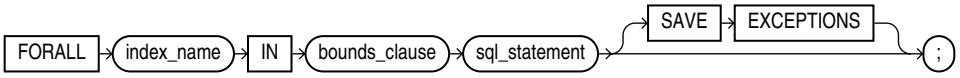
Related Topics

CLOSE Statement, Cursors, Cursor Variables, LOOP Statements, OPEN Statement, OPEN-FOR Statement

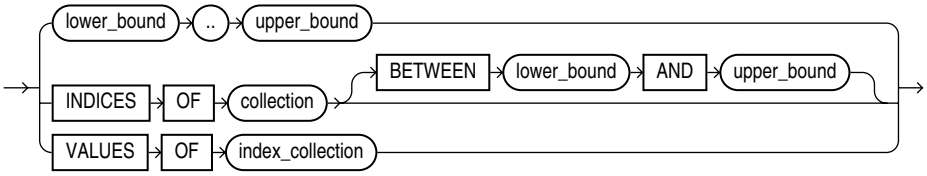
FORALL Statement

The FORALL statement issues a series of INSERT, UPDATE, or DELETE statements, usually much faster than an equivalent FOR loop. It requires some setup code, because each iteration of the loop must use values from one or more collections in its VALUES or WHERE clauses. For more information, see "Reducing Loop Overhead for DML Statements and Queries (FORALL, BULK COLLECT)" on page 11-7.

Syntax



bounds_clause



Keyword and Parameter Description

INDICES OF collection_name

A clause specifying that the values of the index variable correspond to the subscripts of the elements of the specified collection. With this clause, you can use FORALL with nested tables where some elements have been deleted, or with associative arrays that have numeric subscripts.

BETWEEN lower_bound AND upper_bound

Limits the range of subscripts in the INDICES OF clause. If a subscript in the range does not exist in the collection, that subscript is skipped.

VALUES OF index_collection_name

A clause specifying that the subscripts for the FORALL index variable are taken from the values of the elements in another collection, specified by index_collection_name. This other collection acts as a set of pointers; FORALL can iterate through subscripts in arbitrary order, even using the same subscript more than once, depending on what elements you include in index_collection_name.

The index collection must be a nested table, or an associative array indexed by PLS_INTEGER or BINARY_INTEGER, whose elements are also PLS_INTEGER or BINARY_INTEGER. If the index collection is empty, an exception is raised and the FORALL statement is not executed.

index_name

An undeclared identifier that can be referenced only within the FORALL statement and only as a collection subscript.

The implicit declaration of `index_name` overrides any other declaration outside the loop. You cannot refer to another variable with the same name inside the statement. Inside a `FORALL` statement, `index_name` cannot appear in expressions and cannot be assigned a value.

lower_bound .. upper_bound

Numeric expressions that specify a valid range of consecutive index numbers. PL/SQL rounds them to the nearest integer, if necessary. The SQL engine executes the SQL statement once for each index number in the range. The expressions are evaluated once, when the `FORALL` statement is entered.

SAVE EXCEPTIONS

Optional keywords that cause the `FORALL` loop to continue even if some DML operations fail. Instead of raising an exception immediately, the program raises a single exception after the `FORALL` statement finishes. The details of the errors are available after the loop in `SQL%BULK_EXCEPTIONS`. The program can report or clean up all the errors after the `FORALL` loop, rather than handling each exception as it happens.

sql_statement

An `INSERT`, `UPDATE`, or `DELETE` statement that references collection elements in the `VALUES` or `WHERE` clauses.

Usage Notes

Although the SQL statement can reference more than one collection, the performance benefits apply only to subscripted collections.

If a `FORALL` statement fails, database changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous iterations of the `FORALL` loop are *not* rolled back.

Restrictions

The following restrictions apply to the `FORALL` statement:

- You cannot loop through the elements of an associative array that has a string type for the key.
- Within a `FORALL` loop, you cannot refer to the same collection in both the `SET` clause and the `WHERE` clause of an `UPDATE` statement. You might need to make a second copy of the collection and refer to the new name in the `WHERE` clause.
- You can use the `FORALL` statement only in server-side programs, not in client-side programs.
- The `INSERT`, `UPDATE`, or `DELETE` statement must reference at least one collection. For example, a `FORALL` statement that inserts a set of constant values in a loop raises an exception.
- When you specify an explicit range, all collection elements in that range must exist. If an element is missing or was deleted, you get an error.
- When you use the `INDICES OF` or `VALUES OF` clauses, all the collections referenced in the DML statement must have subscripts matching the values of the index variable. Make sure that any `DELETE`, `EXTEND`, and so on operations are applied to all the collections so that they have the same set of subscripts. If any of the collections is missing a referenced element, you get an error. If you use the

SAVE EXCEPTIONS clause, this error is treated like any other error and does not stop the FORALL statement.

- You cannot refer to individual record fields within DML statements called by a FORALL statement. Instead, you can specify the entire record with the SET ROW clause in an UPDATE statement, or the VALUES clause in an INSERT statement.
- Collection subscripts must be just the index variable rather than an expression, such as i rather than i+1.
- The cursor attribute %BULK_ROWCOUNT cannot be assigned to other collections, or be passed as a parameter to subprograms.

Example

You can use the lower and upper 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;
```

Bulk binds apply only to subscripted collections. In the following example, the collection sals, which is passed to the function median, is not bulk-bound:

```
FORALL i IN 1..20
   INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);
```

Related Topics

"Retrieving Query Results into Collections with the BULK COLLECT Clause" on page 11-15

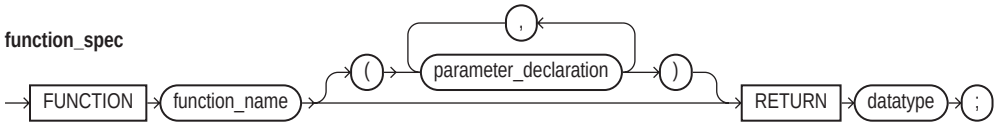
Functions

A function is a subprogram that can take parameters and return a single 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.

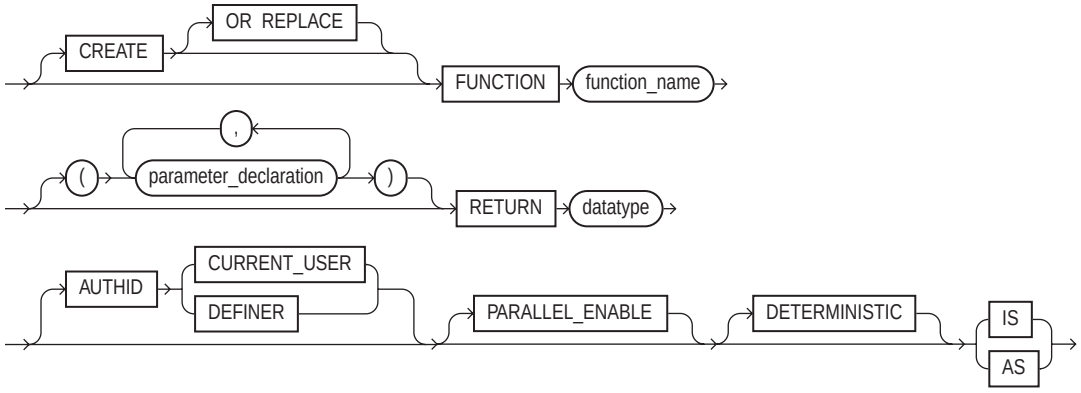
The function body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. These items are local and cease to exist when you exit the function. The executable part contains statements that assign values, control execution, and manipulate data. The exception-handling part contains handlers that deal with exceptions raised during execution. For more information, see "Understanding PL/SQL Functions" on page 8-3.

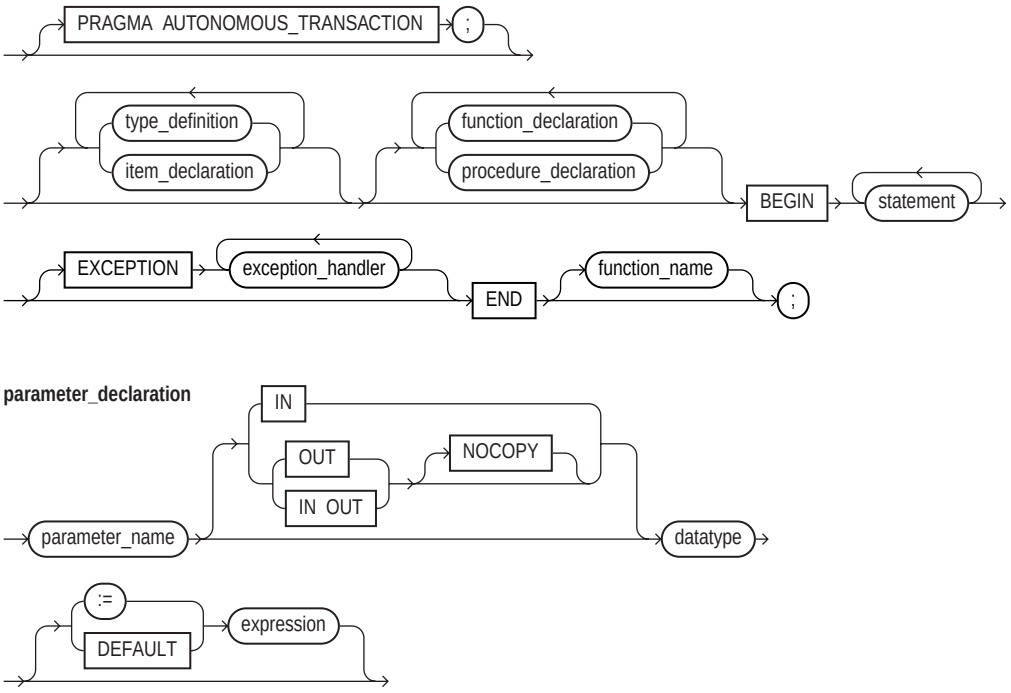
Syntax

function_spec



ffunction_declaration | function body





Keyword and Parameter Description

AUTHID

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 `AUTHID CURRENT_USER`. For more information, see "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

CREATE

The optional `CREATE` clause creates 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

A type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 13-28.

DETERMINISTIC

A hint that 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 *Oracle Database SQL Reference*.

exception_handler

Associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of `exception_handler`, see "Exceptions" on page 13-45.

expression

An arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

function_name

Specifies the name you choose for the function.

IN, OUT, IN OUT

Parameter modes that define the behavior of formal parameters. An `IN` parameter passes values to the subprogram being called. An `OUT` parameter returns values to the caller of the subprogram. An `IN OUT` parameter passes initial values to the subprogram being called, and returns updated values to the caller.

item_declaration

Declares a program object. For its syntax, see "Blocks" on page 13-8.

NOCOPY

A compiler hint (not directive) that allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference instead of by value (the default). The function can run faster, because it does not have to make temporary copies of these parameters, but the results can be different if the function ends with an unhandled exception. For more information, see "Using Default Values for Subprogram Parameters" on page 8-9.

PARALLEL_ENABLE

Declares that a stored function can be used safely in the slave sessions of parallel DML evaluations. The state of a main (logon) session is never shared with slave sessions. Each slave session has its own state, which is initialized when the session begins. The function result should not depend on the state of session (`static`) variables. Otherwise, results might vary across sessions.

parameter_name

A formal parameter, a variable declared in a function spec and referenced in the function body.

PRAGMA AUTONOMOUS_TRANSACTION

Marks a function as *autonomous*. An autonomous transaction is an independent transaction started by 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 6-35.

procedure_declaration

Declares a procedure. For the syntax of `procedure_declaration`, see "Procedures" on page 13-104.

RETURN

Introduces the RETURN clause, which specifies the datatype of the return value.

type_definition

Specifies a user-defined datatype. For its syntax, see "Blocks" on page 13-8.

:= | DEFAULT

Initializes IN parameters to default values.

Usage Notes

A function is called as part of an expression:

```
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

To be callable from SQL statements, a stored function must obey certain rules that control side effects. See "Controlling Side Effects of PL/SQL Subprograms" on page 8-22.

In a function, at least one execution path must lead to a RETURN statement. Otherwise, you get a *function returned without value* error at run time. 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; you cannot assign it a value. An OUT parameter acts like a local variable; you can change its value and reference the value in any way. An IN OUT parameter acts like an initialized variable; you can assign it a value, which can be assigned to another variable. For information about the parameter modes, see Table 8–1 on page 8-8.

Avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take zero or more parameters and return a single value. Functions 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:

```
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

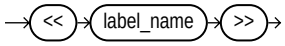
Collection Methods, Packages, Procedures

GOTO Statement

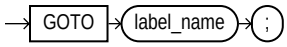
The GOTO statement branches unconditionally to a statement label or block label. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. The GOTO statement transfers control to the labelled statement or block. For more information, see "Using the GOTO Statement" on page 4-12.

Syntax

label_declaration



goto_statement



Keyword and Parameter Description

label_name

A label that you assigned to an executable statement or a PL/SQL block. A GOTO statement transfers control to the statement or block following <<label_name>>.

Usage Notes

Some possible destinations of a GOTO statement are not allowed. In particular, a GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block.

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. To branch to a place that does not have an executable statement, add the NULL statement:

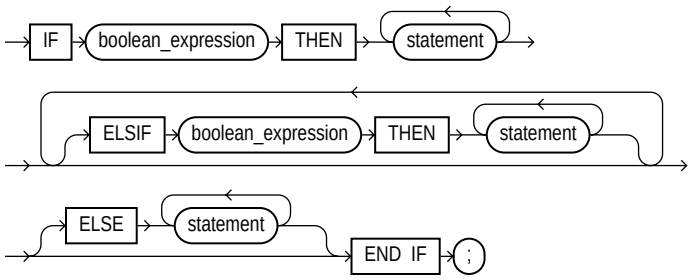
```
FOR ctr IN 1..50 LOOP
   DELETE FROM emp WHERE ...
   IF SQL%FOUND THEN
      GOTO end_loop;
   END IF;
   ...
<<end_loop>>
NULL;   -- an executable statement that specifies inaction
END LOOP;
```

IF Statement

The IF statement executes or skips a sequence of statements, depending on the value of a Boolean expression. For more information, see "Testing Conditions: IF and CASE Statements" on page 4-2.

Syntax

if_statement



Keyword and Parameter Description

boolean_expression

An expression that returns the Boolean value TRUE, FALSE, or NULL. Examples are comparisons for equality, greater-than, or less-than. The sequence following the THEN keyword is executed only if the expression returns TRUE.

ELSE

If control reaches this keyword, the sequence of statements that follows it is executed. This occurs when none of the previous conditional tests returned TRUE.

ELSIF

Introduces a Boolean expression that is evaluated if none of the preceding conditions returned TRUE.

THEN

If the expression returns TRUE, the statements after the THEN keyword are executed.

Usage Notes

There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The simplest form of IF statement associates a Boolean expression with a sequence of statements enclosed by the keywords THEN and END IF. The sequence of statements is executed only if the expression returns TRUE. If the expression returns FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements. The sequence of statements in the ELSE clause is executed only if the Boolean expression returns FALSE or NULL. Thus, the ELSE clause ensures that a sequence of statements is executed.

The third form of IF statement uses the keyword ELSIF to introduce additional Boolean expressions. If the first expression returns FALSE or NULL, the ELSIF clause

evaluates another expression. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Boolean expressions are evaluated one by one from top to bottom. If any expression returns TRUE, its associated sequence of statements is executed and control passes to the next statement. If all expressions return FALSE or NULL, the sequence in the ELSE clause is executed.

An IF statement never executes more than one sequence of statements because processing is complete after any sequence of statements is executed. However, the THEN and ELSE clauses can include more IF statements. That is, IF statements can be nested.

Examples

In the example below, if shoe_count has a value of 10, both the first and second Boolean expressions return TRUE. Nevertheless, order_quantity is assigned the proper value of 50 because processing of an IF statement stops after an expression returns TRUE and its associated sequence of statements is executed. The expression associated with ELSIF is never evaluated and control passes to the INSERT statement.

```
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:

```
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;
```

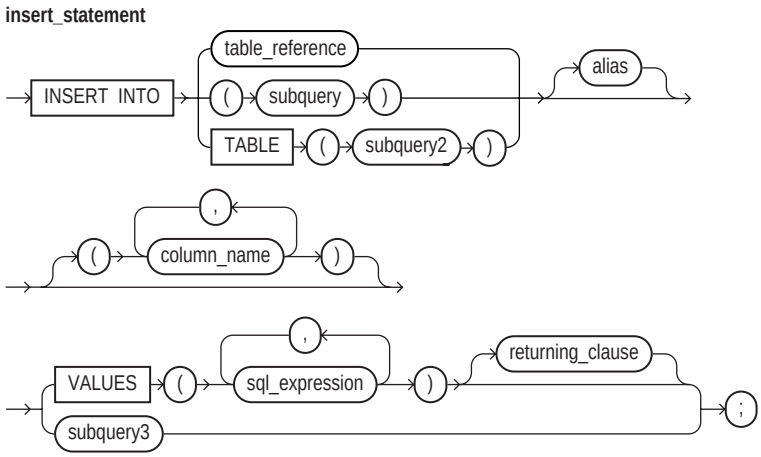
Related Topics

CASE Statement, Expressions

INSERT Statement

The `INSERT` statement adds one or more new rows of data to a database table. For a full description of the `INSERT` statement, see *Oracle Database SQL Reference*.

Syntax



Keyword and Parameter Description

alias

Another (usually short) name for the referenced table or view.

column_name[, column_name]...

A list of columns in a database table or view. The columns can be listed in any order, as long as the expressions in the `VALUES` clause are listed in the same order. Each column name can only be listed once. If the list does not include all the columns in a table, each missing columns is set to `NULL` or to a default value specified in the `CREATE TABLE` statement.

returning_clause

Returns values from inserted rows, eliminating the need to `SELECT` the rows afterward. You can retrieve the column values into variables or into collections. You cannot use the `RETURNING` clause for remote or parallel inserts. If the statement does not affect any rows, the values of the variables specified in the `RETURNING` clause are undefined. For the syntax of `returning_clause`, see "DELETE Statement" on page 13-41.

sql_expression

Any expression valid in SQL. For example, it could be a literal, a PL/SQL variable, or a SQL query that returns a single value. For more information, see *Oracle Database SQL Reference*. PL/SQL also lets you use a record variable here.

subquery

A `SELECT` statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 13-123.

subquery3

A SELECT statement that returns a set of rows. Each row returned by the select statement is inserted into the table. The subquery must return a value for every column in the column list, or for every column in the table if there is no column list.

table_reference

A table or view that must be accessible when you execute the INSERT statement, and for which you must have INSERT privileges. For the syntax of table_reference, see "DELETE Statement" on page 13-41.

TABLE (subquery2)

The operand of TABLE is a SELECT statement that returns a single column value representing a nested table. This operator specifies that the value is a collection, not a scalar value.

VALUES (...)

Assigns the values of expressions to corresponding columns in the column list. If there is no column list, the first value is inserted into the first column defined by the CREATE TABLE statement, the second value is inserted into the second column, and so on. There must be one value for each column in the column list. The datatypes of the values being inserted must be compatible with the datatypes of corresponding columns in the column list.

Usage Notes

Character and date literals in the VALUES list must be enclosed by single quotes ('). Numeric literals are not enclosed by quotes.

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of an INSERT statement.

Examples

The following examples show various forms of INSERT statement:

```
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

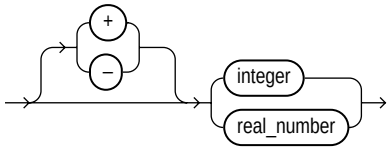
DELETE Statement, SELECT INTO Statement, UPDATE Statement

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 2-4.

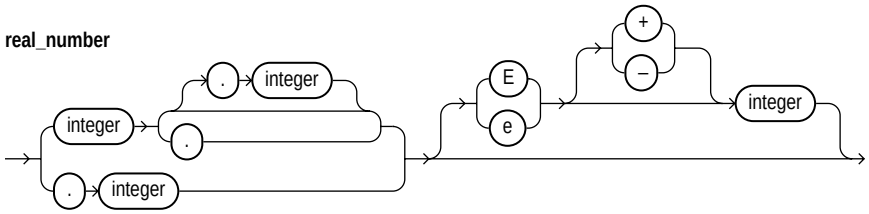
Syntax

numeric_literal

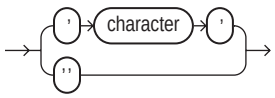


integer

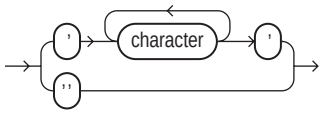
real_number



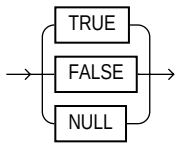
character_literal



string_literal



boolean_literal



Keyword and Parameter Description

character

A member of the PL/SQL character set. For more information, see "Character Set" on page 2-1.

digit

One of the numerals 0 .. 9.

TRUE, FALSE, NULL

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. 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. A string literal can hold up to 32,767 characters.

To represent an apostrophe within a string, enter two single quotes instead of one. For literals where doubling the quotes is inconvenient or hard to read, you can designate an escape character using the notation `q'`*esc_char* `...` *esc_char*`'`. This escape character must not occur anywhere else inside the string.

PL/SQL is case sensitive within string literals. For example, PL/SQL considers the literals `'white'` and `'White'` to be different.

Trailing blanks are significant within string literals, so `'abc'` and `'abc '` are different. Trailing blanks in a string literal are not trimmed during PL/SQL processing, although they are trimmed if you insert that value into a table column of type `CHAR`.

The Boolean values `TRUE` and `FALSE` cannot be inserted into a database column.

Examples

Several examples of numeric literals are:

```
25   6.34   7E2   25e-03   .1   1.   +17  -4.4  -4.5D  -4.6F
```

Several examples of character literals are:

```
'H'   '&'   ' '   '9'   ']'   'g'
```

Several examples of string literals are:

```
'$5,000'
'02-AUG-87'
'Don''t leave until you''re ready and I''m ready.'
q'#Don't leave until you're ready and I'm ready.#'
```

Related Topics

Constants and Variables, Expressions

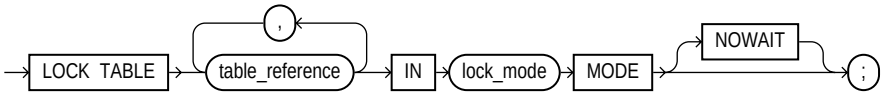
LOCK TABLE Statement

The `LOCK TABLE` statement locks entire database tables in a specified lock mode. That enables you to share or deny access to tables while maintaining their integrity. For more information, see "Using LOCK TABLE" on page 6-33.

Oracle has extensive automatic features that allow multiple programs to read and write data simultaneously, while each program sees a consistent view of the data; you should rarely, if ever, need to lock tables yourself.

Syntax

lock_table_statement



Keyword and Parameter Description

table_reference

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 13-41.

lock_mode

The type of lock. 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

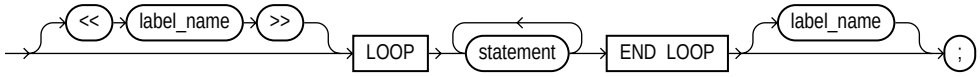
COMMIT Statement, ROLLBACK Statement

LOOP Statements

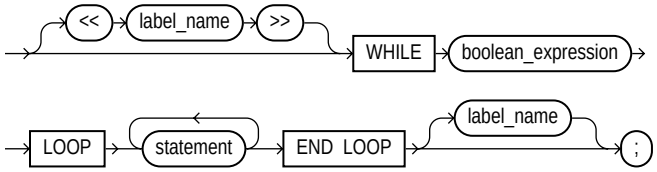
`LOOP` statements execute a sequence of statements multiple times. The LOOP and END LOOP keywords enclose the statements. PL/SQL provides four kinds of loop statements: basic loop, `WHILE` loop, `FOR` loop, and cursor `FOR` loop. For usage information, see "Controlling Loop Iterations: LOOP and EXIT Statements" on page 4-6.

Syntax

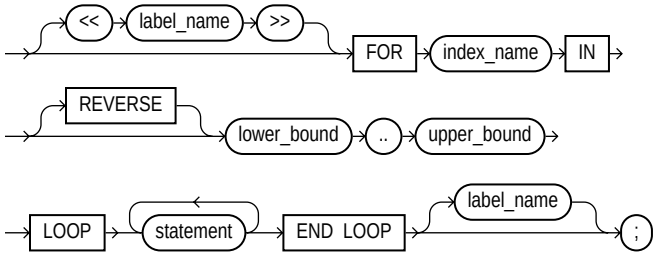
basic_loop_statement



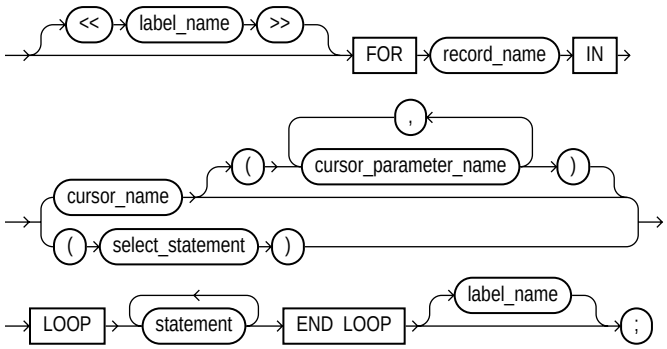
while_loop_statement



for_loop_statement



cursor_for_loop_statement



Keyword and Parameter Description

basic_loop_statement

A loop that executes an unlimited number of times. It encloses a sequence of statements between the keywords `LOOP` and `END LOOP`. With each iteration, the sequence of statements is executed, then control resumes at the top of the loop. An `EXIT`, `GOTO`, or `RAISE` statement branches out of the loop. A raised exception also ends the loop.

boolean_expression

An expression that returns the Boolean value `TRUE`, `FALSE`, or `NULL`. It is associated with a sequence of statements, which is executed only if the expression returns `TRUE`. For the syntax of `boolean_expression`, see "Expressions" on page 13-52.

cursor_for_loop_statement

Issues a SQL query and loops through the rows in the result set. This is a convenient technique that makes processing a query as simple as reading lines of text in other programming languages.

A cursor `FOR` loop implicitly declares its loop index as a `%ROWTYPE` record, opens a cursor, repeatedly fetches rows of values from the result set into fields in the record, and closes the cursor when all rows have been processed.

cursor_name

An explicit cursor previously declared within the current scope. When the cursor `FOR` loop is entered, `cursor_name` cannot refer to a cursor already opened by an `OPEN` statement or an enclosing cursor `FOR` loop.

cursor_parameter_name

A variable declared as the formal parameter of a cursor. (For the syntax of `cursor_parameter_declaration`, see "Cursors" on page 13-38.) A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be `IN` parameters.

for_loop_statement

Numeric `FOR` loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords `FOR` and `LOOP`.

The range is evaluated when the `FOR` loop is first entered and is never re-evaluated. The loop body is executed once for each integer in the range defined by `lower_bound..upper_bound`. After each iteration, the loop index is incremented.

index_name

An undeclared identifier that names the loop index (sometimes called a loop counter). Its scope is the loop itself; you cannot reference the index outside the loop.

The implicit declaration of `index_name` overrides any other declaration outside the loop. To refer to another variable with the same name, use a label:

```
<<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, the index is treated like a constant: it can appear in expressions, but cannot be assigned a value.

label_name

An optional undeclared identifier that labels a loop. `label_name` must be enclosed by double angle brackets and must appear at the beginning of the loop. Optionally, `label_name` (*not* enclosed in angle brackets) can also appear at the end of the loop.

You can use `label_name` in an `EXIT` statement to exit the loop labelled by `label_name`. You can exit not only the current loop, but any enclosing loop.

You cannot reference the index of a `FOR` loop from a nested `FOR` loop if both indexes have the same name, unless the outer loop is labeled by `label_name` and you use dot notation:

```
label_name.index_name
```

The following example compares 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

Expressions that return numbers. (Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`.) The expressions are evaluated only when the loop is first entered. The lower bound need not be 1, as the example below shows. 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`. 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

An implicitly declared record. The record has the same structure as a row retrieved by `cursor_name` or `select_statement`.

The record is defined only inside the loop. You cannot refer to its fields outside the loop. The implicit declaration of `record_name` overrides any other declaration outside the loop. You cannot refer to another record with the same name inside the loop unless you qualify the reference using a block label.

Fields in the record store column values from the implicitly fetched row. The fields have the same names and datatypes as their corresponding columns. To access field values, you use dot notation, as follows:

```
record_name.field_name
```

Select-items fetched from the `FOR` loop cursor must have simple names or, if they are expressions, must have aliases. In the following example, `wages` is an alias for the select item `sal+NVL(comm,0)`:

```
CURSOR c1 IS SELECT empno, sal+comm wages, job ...
```

REVERSE

By default, iteration proceeds upward from the lower bound to the upper bound. If you use the keyword `REVERSE`, iteration proceeds downward from the upper bound to the lower bound.

An example follows:

```
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

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 13-123. PL/SQL automatically declares, opens, fetches from, and closes the internal cursor. Because `select_statement` is not an independent statement, the implicit cursor `SQL` does not apply to it.

while_loop_statement

The `WHILE-LOOP` statement associates a Boolean expression with a sequence of statements enclosed by the keywords `LOOP` and `END LOOP`. Before each iteration of the loop, the expression is evaluated. If the expression returns `TRUE`, the sequence of statements is executed, then control resumes at the top of the loop. If the expression returns `FALSE` or `NULL`, the loop is bypassed and control passes to the next statement.

Usage Notes

You can use the `EXIT WHEN` statement to exit any loop prematurely. If the Boolean expression in the `WHEN` clause returns `TRUE`, the loop is exited immediately.

When you exit a cursor FOR loop, the cursor is closed automatically even if you use an EXIT or GOTO statement to exit the loop prematurely. The cursor is also closed automatically if an exception is raised inside the loop.

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 a full description and examples of the `MERGE` statement, see *Oracle Database SQL Reference*.

Usage Notes

This statement is primarily useful in data warehousing situations where large amounts of data are commonly inserted and updated. If you only need to insert or update a single row, it is more efficient to do that with the regular PL/SQL techniques: try to update the row, and do an insert instead if the update affects zero rows; or try to insert the row, and do an update instead if the insert raises an exception because the table already contains that primary key.

NULL Statement

The NULL statement is a *no-op*: it passes control to the next statement without doing anything. In the body of an IF-THEN clause, a loop, or a procedure, the NULL statement serves as a placeholder. For more information, see "Using the NULL Statement" on page 4-13.

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: you have decided that no action is necessary.

Certain clauses in PL/SQL, such as in an IF statement or an exception handler, must contain at least one executable statement. You can use the NULL statement to make these constructs compile, while not taking any action.

You might not be able to branch to certain places with the GOTO statement because the next statement is END, END IF, and so on, which are not executable statements. In these cases, you can put a NULL statement where you want to branch.

The NULL statement and Boolean value NULL are not related.

Examples

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;
```

Object Types

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**. A special kind of method called the **constructor** creates a new instance of the object type and fills in its attributes.

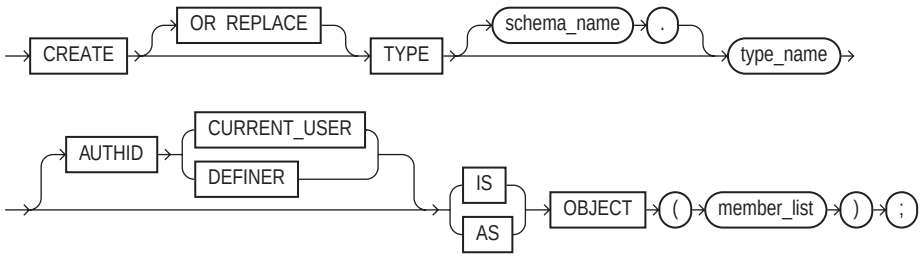
Object types must be created through SQL and stored in an Oracle database, where they can be shared by many programs. When you define an object type using the CREATE TYPE statement, you create an abstract template for some real-world object. The template specifies the attributes and behaviors the object needs 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, so that the data is kept in a proper state.

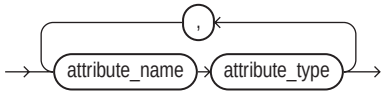
For information on using object types, see Chapter 12, "Using PL/SQL Object Types".

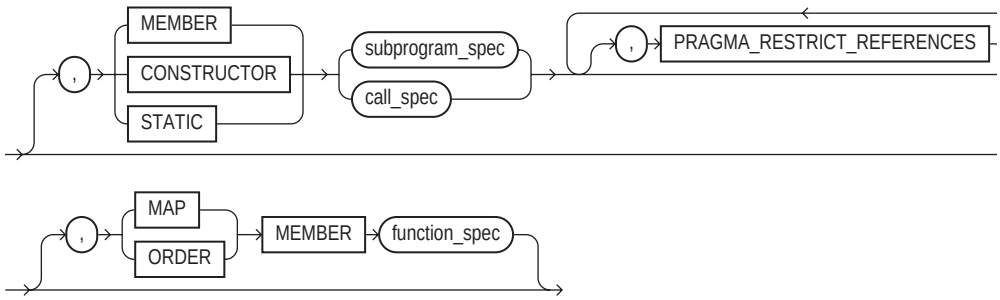
Syntax

object_type_declaration | object_type_spec

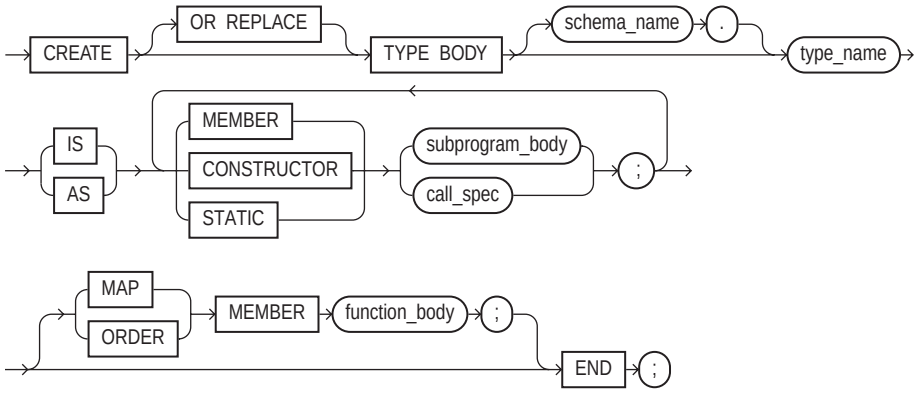


member_list





object_type_body



Keyword and Parameter Description

attribute_datatype

Any Oracle datatype except `LONG`, `LONG RAW`, `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

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. You cannot impose the `NOT NULL` constraint on an attribute.

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 "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

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 *Oracle Database Java Developer's Guide*. To learn how to write C call specs *Oracle Database Application Developer's Guide - Fundamentals*.

function_body

Implements a CONSTRUCTOR, MEMBER, or STATIC function. For the syntax of function_body, see "Functions" on page 13-67.

MAP

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 | CONSTRUCTOR | STATIC

Declares a subprogram or call spec as a method in an object type spec. A constructor method must have the same name as the object type, while member and static methods must have names that are different from the object type or any of its attributes.

MEMBER methods are invoked on instances of objects, and read or change the attributes of that particular instance:

```
object_instance.method();
```

CONSTRUCTOR methods create new instances of objects, and fill in some or all of the attributes:

```
object_instance := new object_type_name(attr1 => attr1_value,
  attr2 => attr2_value);
```

The system defines a default constructor method with one parameter for each object attribute, so you only need to define your own constructor methods if you want to construct the object based on a different set of parameters.

STATIC methods are invoked on the object type, not any specific object instance, and thus must limit themselves to "global" operations that do not involve the object attributes:

```
object_type.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. The headers must match word for word. Differences in whitespace are allowed.

CONSTRUCTOR and 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 constructor functions, SELF always has the parameter mode IN OUT. In MEMBER functions, if SELF is not declared, its parameter mode defaults to IN. 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

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 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.

pragma_restrict_refs

Pragma RESTRICT_REFERENCES, which checks 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 13-113 (in this context, omit the pragma terminator).

The pragma asserts that a member function does not read or write database tables or package variables. For more information about the purity rules and pragma RESTRICT_REFERENCES, see *Oracle Database Application Developer's Guide - Fundamentals*.

schema_name

The schema containing the object type. If you omit schema_name, Oracle assumes the object type is in your schema.

subprogram_body

Implements 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 13-67 and/or "Procedures" on page 13-104.

subprogram_spec

Declares the interface to a CONSTRUCTOR, MEMBER or STATIC function or procedure. Its syntax is like that of function_spec or procedure_spec without the terminator. See "Functions" on page 13-67 and/or "Procedures" on page 13-104.

type_name

A user-defined object type that was defined using the datatype specifier OBJECT.

Usage Notes

Once an object type is created in your schema, you can use it to declare objects in any PL/SQL block, subprogram, or package. For example, you can use the object type to specify the datatype of an object attribute, table column, PL/SQL variable, bind variable, record field, collection element, formal procedure parameter, or function result.

Like a package, an object type has two parts: a specification and a body. The specification (spec for short) is the interface to your applications; it declares a data structure (set of attributes) along with the operations (methods) needed to manipulate the data. The body fully defines the methods, and so implements the spec.

All the information a client program needs to use the methods is in the spec. Think of the spec as an operational interface and of the body as a black box. You can debug, enhance, or replace the body without changing the spec.

An object type encapsulates data and operations. You can declare attributes and methods in an object type spec, but *not* constants, exceptions, cursors, or types. At least one attribute is required (the maximum is 1000); methods are optional.

In an object type spec, all attributes must be declared before any methods. Only subprograms have an underlying implementation. If an object type 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 default constructor method (constructor for short), which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type. You can also define your own constructor methods that accept different sets of parameters. PL/SQL never calls a constructor implicitly, so you must call it explicitly. Constructor calls are allowed wherever function calls are allowed.

Examples

This SQL*Plus script defines an object type for a stack. 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.

```
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:

```
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:

```
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,
);
```

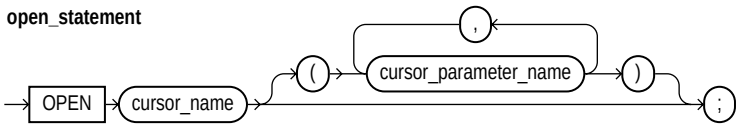
Related Topics

Functions, Packages, Procedures

OPEN Statement

The OPEN statement executes the query associated with a cursor. It allocates database resources to process the query and identifies the result set -- the rows that match the query conditions. The cursor is positioned before the first row in the result set. For more information, see "Querying Data with PL/SQL" on page 6-9.

Syntax



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope and not currently open.

cursor_parameter_name

A variable declared as the formal parameter of a cursor. (For the syntax of cursor_parameter_declaration, see "Cursors" on page 13-38.) A cursor parameter can appear in a query wherever a constant can appear.

Usage Notes

Generally, PL/SQL parses an explicit cursor only the first time it is opened and parses a SQL statement (creating an implicit cursor) only the first time the statement is executed. All the parsed SQL statements are cached. A SQL statement is reparsed only if it is aged out of the cache by a new SQL statement. Although you must close a cursor before you can reopen it, PL/SQL need not reparse the associated SELECT statement. If you close, then immediately reopen the cursor, a reparse is definitely not needed.

Rows in the result set are not retrieved when the OPEN statement is executed. The FETCH statement retrieves the rows. With a FOR UPDATE cursor, the rows are locked when the cursor is opened.

If formal parameters are declared, actual parameters must be passed to the cursor. The formal parameters of a cursor must be IN parameters; they cannot return values to actual parameters. The values of actual parameters are used when the cursor is opened. The datatypes of the formal and actual parameters must be compatible. The query can also reference PL/SQL variables declared within its scope.

Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. Formal parameters declared with a default value do not need a corresponding actual parameter. They assume their default values when the OPEN statement is executed.

You can associate the actual parameters in an OPEN statement with the formal parameters in a cursor declaration using positional or named notation.

If a cursor is currently open, you cannot use its name in a cursor FOR loop.

Examples

Given the cursor declaration:

```
CURSOR parts_cur IS SELECT part_num, part_price FROM parts;
```

the following statement opens the cursor:

```
OPEN parts_cur;
```

Given the cursor declaration:

```
CURSOR emp_cur(my_ename VARCHAR2, my_comm NUMBER DEFAULT 0)
   IS SELECT * FROM emp WHERE ...
```

any of the following statements opens the cursor:

```
OPEN emp_cur('LEE');
OPEN emp_cur('BLAKE', 300);
OPEN emp_cur(employee_name, 150);
```

Related Topics

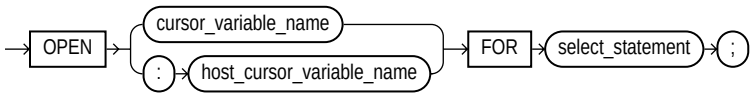
CLOSE Statement, Cursors, FETCH Statement, LOOP Statements

OPEN-FOR Statement

The `OPEN-FOR` statement executes the query associated with a cursor variable. It allocates database resources to process the query and identifies the result set -- the rows that meet the query conditions. The cursor variable is positioned before the first row in the result set. For more information, see "Using Cursor Variables (REF CURSORs)" on page 6-19.

Syntax

open_for_statement



Keyword and Parameter Description

cursor_variable_name

A cursor variable (or parameter) previously declared within the current scope.

host_cursor_variable_name

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

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 13-123, except that the cursor `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:

```
/* 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 do not need to close a cursor variable before reopening it. When you reopen a cursor variable for a different query, the previous query is lost.

Unlike cursors, cursor variables do not take parameters. Instead, you can pass whole queries (not just parameters) to a cursor variable.

Although a PL/SQL stored procedure or function can open a cursor variable and pass it back to a calling subprogram, the calling and called subprograms must be in the same instance. You cannot pass or return cursor variables to procedures and functions called through database links.

When you declare a cursor variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the `IN OUT` mode. That way, the subprogram can pass an open cursor back to the caller.

Examples

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:

```
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:

```
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

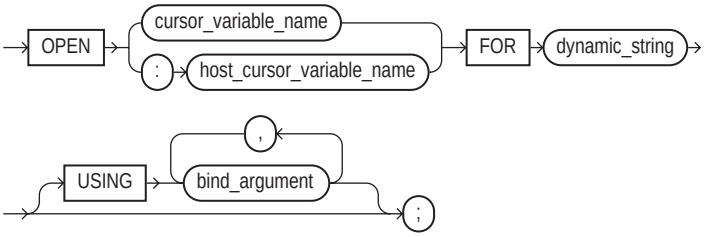
OPEN-FOR-USING Statement

The `OPEN-FOR-USING` statement associates a cursor variable with a 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 "Building a Dynamic Query with Dynamic SQL" on page 7-4.

Because this statement can use bind variables to make the SQL processing more efficient, use the `OPEN-FOR-USING` statement when building a query where you know the `WHERE` clauses in advance. Use the `OPEN-FOR` statement when you need the flexibility to process a dynamic query with an unknown number of `WHERE` clauses.

Syntax

open_for_using_statement



Keyword and Parameter Description

cursor_variable_name

A weakly typed cursor variable (one without a return type) previously declared within the current scope.

bind_argument

An expression whose value is passed to the dynamic `SELECT` statement.

dynamic_string

A string literal, variable, or expression that represents a multi-row `SELECT` statement.

host_cursor_variable_name

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 to Dynamic SQL" on page 7-10.

Any bind arguments in the query are evaluated only when the cursor variable is opened. 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

The following example declares a cursor variable, then associates it with a dynamic SELECT statement:

```
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

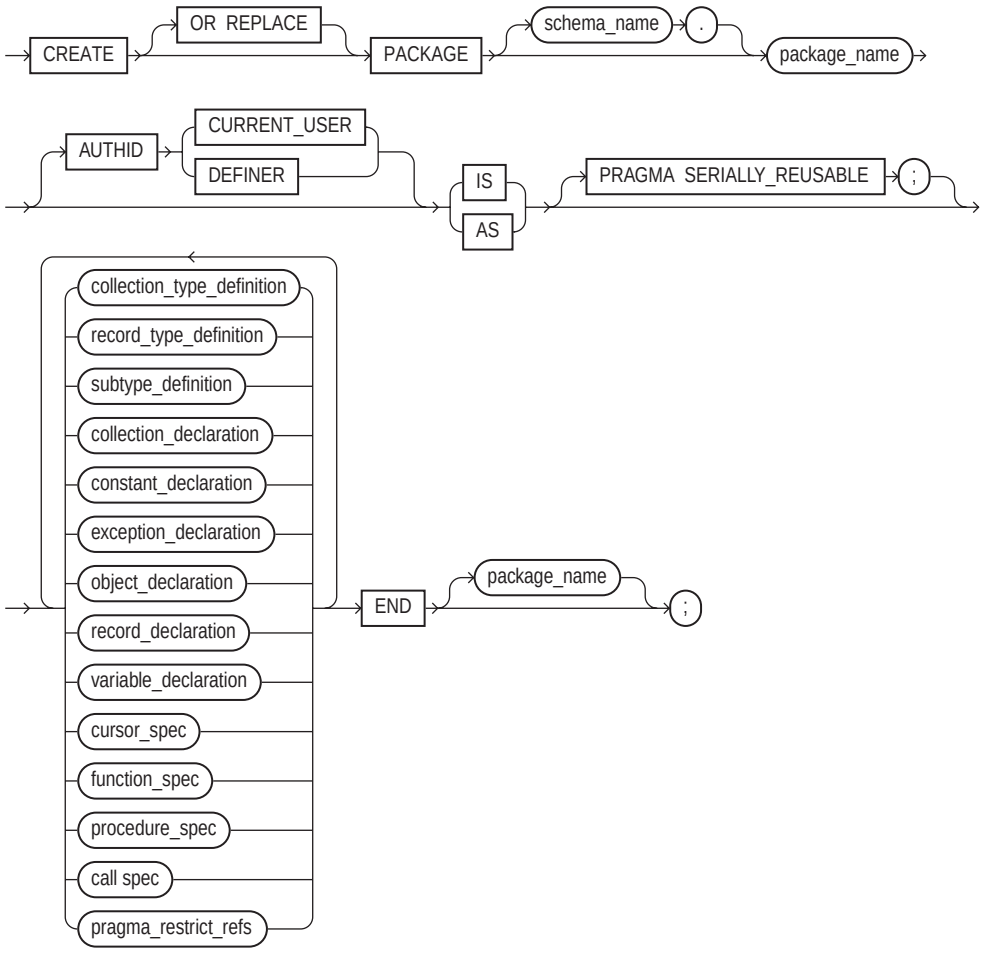
EXECUTE IMMEDIATE Statement, OPEN-FOR Statement

Packages

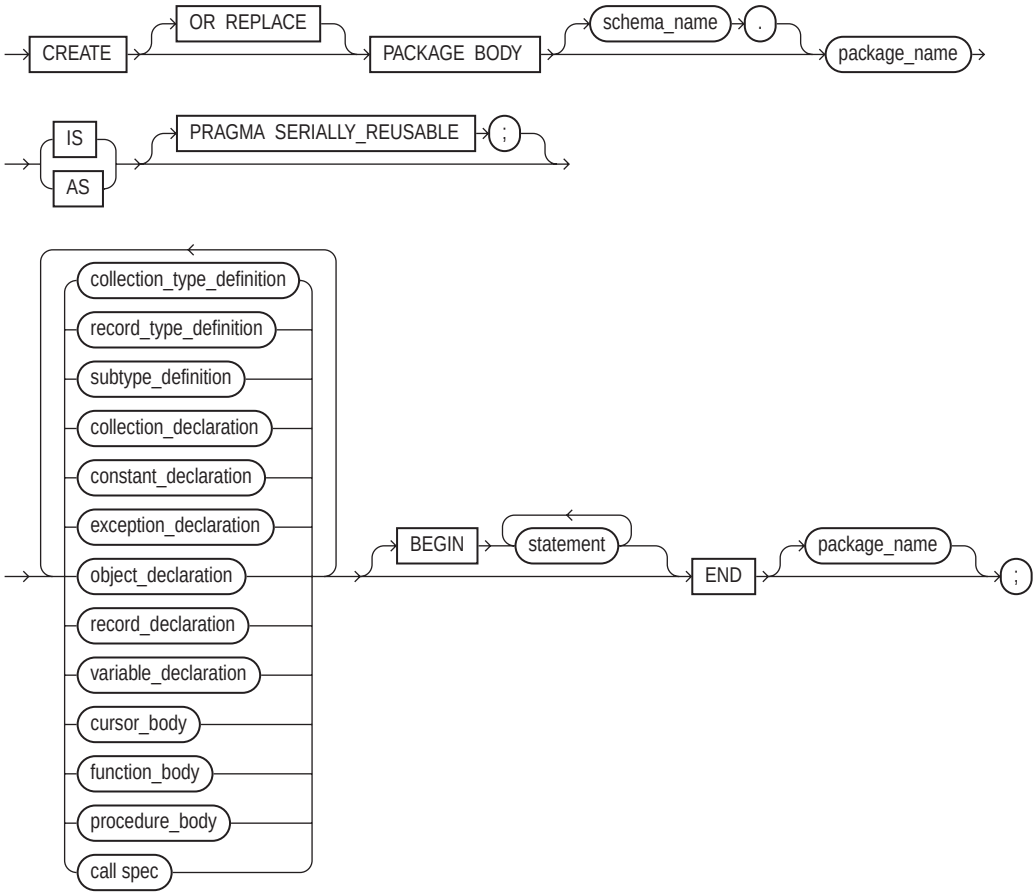
A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Use packages when writing a set of related subprograms that form an application programming interface (API) that you or others might reuse. Packages have two parts: a specification (spec for short) and a body. For more information, see Chapter 9, "Using PL/SQL Packages".

Syntax

package_declaration | package_spec



package_body



Keyword and Parameter Description

AUTHID

Determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

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. For more information, see *Oracle Database Java Developer's Guide* and *Oracle Database Application Developer's Guide - Fundamentals*.

collection_declaration

Declares a collection (nested table, index-by table, or varray). For the syntax of `collection_declaration`, see "Collections" on page 13-21.

collection_type_definition

Defines a collection type using the datatype specifier TABLE or VARRAY.

constant_declaration

Declares a constant. For the syntax of `constant_declaration`, see "Constants and Variables" on page 13-28.

cursor_body

Defines the underlying implementation of an explicit cursor. For the syntax of `cursor_body`, see "Cursors" on page 13-38.

cursor_spec

Declares the interface to an explicit cursor. For the syntax of `cursor_spec`, see "Cursors" on page 13-38.

exception_declaration

Declares an exception. For the syntax of `exception_declaration`, see "Exceptions" on page 13-45.

function_body

Implements a function. For the syntax of `function_body`, see "Functions" on page 13-67.

function_spec

Declares the interface to a function. For the syntax of `function_spec`, see "Functions" on page 13-67.

object_declaration

Declares an object (instance of an object type). For the syntax of `object_declaration`, see "Object Types" on page 13-86.

package_name

A package stored in the database. For naming conventions, see "Identifiers" on page 2-3.

pragma_restrict_refs

Pragma `RESTRICT_REFERENCES`, which checks for violations of "purity" rules. To be callable from SQL statements, a function must obey rules that 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 13-113.

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 *Oracle Database Application Developer's Guide - Fundamentals*.

PRAGMA SERIALLY_REUSABLE

Marks a package as *serially reusable*, 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 remote procedure call). For more information, see *Oracle Database Application Developer's Guide - Fundamentals*.

procedure_body

Implements a procedure. For the syntax of `procedure_body`, see "Procedures" on page 13-104.

procedure_spec

Declares the interface to a procedure. For the syntax of `procedure_spec`, see "Procedures" on page 13-104.

record_declaration

Declares a user-defined record. For the syntax of `record_declaration`, see "Records" on page 13-110.

record_type_definition

Defines a record type using the datatype specifier `RECORD` or the attribute `%ROWTYPE`.

schema_name

The schema containing the package. If you omit `schema_name`, Oracle assumes the package is in your schema.

variable_declaration

Declares a variable. For the syntax of `variable_declaration`, see "Constants and Variables" on page 13-28.

Usage Notes

You can use any Oracle tool that supports PL/SQL to create and store packages in an Oracle database. You can issue the `CREATE PACKAGE` and `CREATE PACKAGE BODY` statements interactively from SQL*Plus, or from an Oracle Precompiler or OCI host program.

You cannot define packages in a PL/SQL block or subprogram.

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. If a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. The body can still be used to initialize items declared in the spec:

```
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 do not need to define the package bodies fully until you are ready to complete the application. You can debug, enhance, or replace a package body without changing the package spec, which saves you from recompiling subprograms that call the package.

Cursors and subprograms declared in a package spec must be defined in the package body. Other program items declared in the package spec cannot be redeclared in the package body.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. Except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception.

Variables declared in a package keep their values throughout a session, so you can set the value of a package variable in one procedure, and retrieve the same value in a different procedure.

Related Topics

Collections, Cursors, Exceptions, Functions, Procedures, Records

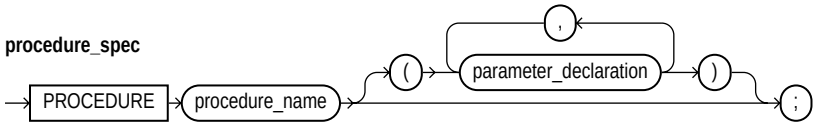
Procedures

A procedure is a subprogram that can take parameters and be called. 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.

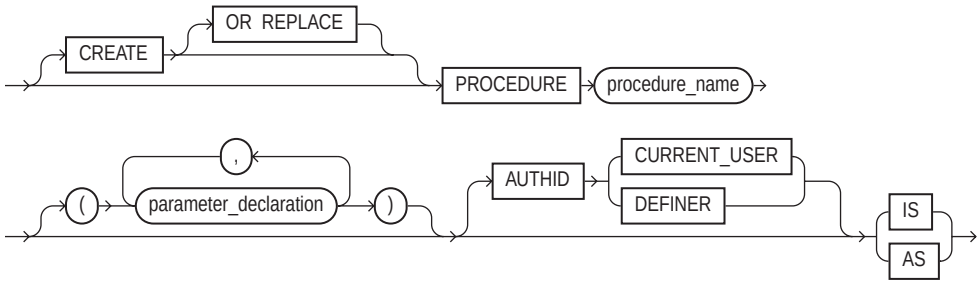
The procedure body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. These items are local and cease to exist when you exit the procedure. 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.

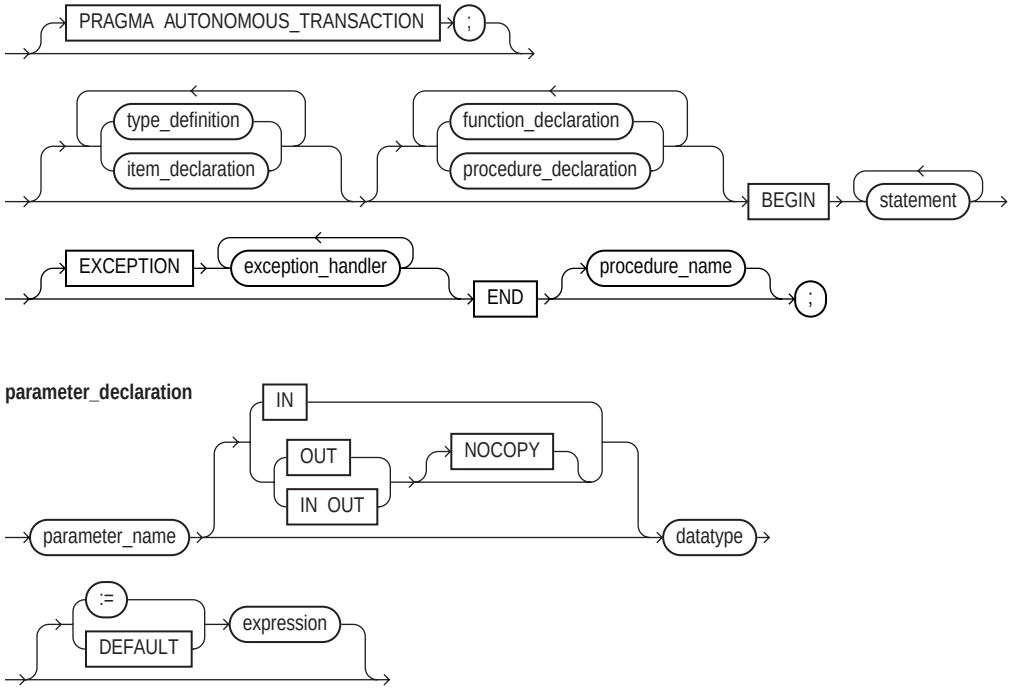
Syntax

procedure_spec



procedure_declaration | procedure body





Keyword and Parameter Description

AUTHID

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 "Using Invoker's Rights Versus Definer's Rights (AUTHID Clause)" on page 8-15.

CREATE

The optional `CREATE` clause creates stored procedures, which are stored in the Oracle database and can be called from other applications. You can execute the `CREATE` statement interactively from SQL*Plus or from a program using native dynamic SQL.

datatype

A type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 13-28.

exception_handler

Associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of `exception_handler`, see "Exceptions" on page 13-45.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

function_declaration

Declares a function. For the syntax of `function_declaration`, see "Functions" on page 13-67.

IN, OUT, IN OUT

Parameter modes that define the behavior of formal parameters. An `IN` parameter passes values to the subprogram being called. An `OUT` parameter returns values to the caller of the subprogram. An `IN OUT` parameter lets passes initial values to the subprogram being called and returns updated values to the caller.

item_declaration

Declares a program object. For the syntax of `item_declaration`, see "Blocks" on page 13-8.

NOCOPY

A compiler hint (not directive) that allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference instead of by value (the default). For more information, see "Specifying Subprogram Parameter Modes" on page 8-7.

parameter_name

A formal parameter, which is a variable declared in a procedure spec and referenced in the procedure body.

PRAGMA AUTONOMOUS_TRANSACTION

Marks a function as *autonomous*. An autonomous transaction is an independent transaction started by 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 6-35.

procedure_name

A user-defined procedure.

type_definition

Specifies a user-defined datatype. For the syntax of `type_definition`, see "Blocks" on page 13-8.

:= | DEFAULT

Initializes `IN` parameters to default values, if they are not specified when the procedure is called.

Usage Notes

A procedure is called as a PL/SQL statement. For example, the procedure `raise_salary` might be called as follows:

```
raise_salary(emp_num, amount);
```

Inside a procedure, an `IN` parameter acts like a constant; you cannot assign it a value. An `OUT` parameter acts like a local variable; you can change its value and reference the value in any way. An `IN OUT` parameter acts like an initialized variable; you can assign it a value, which can be assigned to another variable. For summary information about the parameter modes, see Table 8–1 on page 8-8.

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 8-9.

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:

```
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;
```

The following example calls the procedure using named notation:

```
debit_account(amount => 500, acct_id => 10261);
```

Related Topics

Collection Methods, Functions, Packages

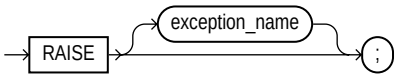
RAISE Statement

The RAISE statement stops normal execution of a PL/SQL block or subprogram and transfers control to an exception handler.

RAISE statements can raise predefined exceptions, such as ZERO_DIVIDE or NO_DATA_FOUND, or user-defined exceptions whose names you decide. For more information, see "Defining Your Own PL/SQL Exceptions" on page 10-6.

Syntax

raise_statement



Keyword and Parameter Description

exception_name

A predefined or user-defined exception. For a list of the predefined exceptions, see "Summary of Predefined PL/SQL Exceptions" on page 10-4.

Usage Notes

PL/SQL blocks and subprograms should RAISE an exception only when an error makes it impractical to continue processing. You can code a RAISE statement for a given exception anywhere within the scope of that exception.

When an exception is raised, if PL/SQL cannot find a handler for it in the current block, the exception propagates to successive enclosing blocks, until a handler is found or there are no more blocks to search. If no handler is found, PL/SQL returns an *unhandled exception* error to the host environment.

In an exception handler, you can omit the exception name in a RAISE statement, which raises the current exception again. This technique allows you to take some initial corrective action (perhaps just logging the problem), then pass control to another handler that does more extensive correction. When an exception is reraised, the first block searched is the enclosing block, not the current block.

Example

The following example raises an exception when an inventoried part is out of stock, or when a divide-by-zero situation is about to occur:

```
DECLARE
   out_of_stock EXCEPTION;
   quantity_on_hand NUMBER := 0;
   denominator NUMBER := 0;
BEGIN
   IF quantity_on_hand = 0 THEN
      RAISE out_of_stock;
   END IF;

   IF denominator = 0 THEN
      raise ZERO_DIVIDE;
   END IF;
```

```
   EXCEPTION
      WHEN out_of_stock THEN
         dbms_output.put_line('No more parts in stock.');
      WHEN ZERO_DIVIDE THEN
         dbms_output.put_line('Attempt to divide by zero.');
      WHEN OTHERS THEN
         dbms_output.put_line('Some other kind of problem...');
END;
/
```

Related Topics

Exceptions

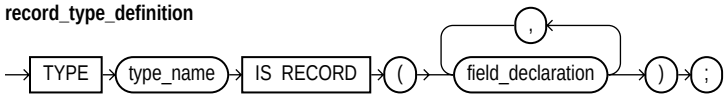
Records

Records are composite variables that can store data values of different types, similar to a `struct` type in C, C++, or Java. For more information, see "What Is a PL/SQL Record?" on page 5-32.

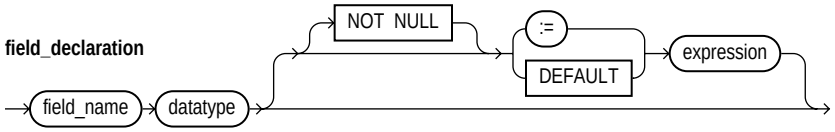
In PL/SQL records are useful for holding data from table rows, or certain columns from table rows. For ease of maintenance, you can declare variables as table%ROWTYPE or cursor%ROWTYPE instead of creating new record types.

Syntax

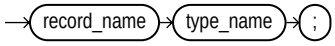
record_type_definition



field_declaration



record_declaration



Keyword and Parameter Description

datatype

A datatype specifier. For the syntax of `datatype`, see "Constants and Variables" on page 13-28.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of `expression`, see "Expressions" on page 13-52. When the declaration is elaborated, the value of `expression` is assigned to the field. The value and the field must have compatible datatypes.

field_name

A field in a user-defined record.

NOT NULL

At run time, trying to assign a null to a field defined as `NOT NULL` raises the predefined exception `VALUE_ERROR`. The constraint `NOT NULL` must be followed by an initialization clause.

record_name

A user-defined record.

type_name

A user-defined record type that was defined using the datatype specifier RECORD.

:= | DEFAULT

Initializes fields to default values.

Usage Notes

You can define RECORD types and declare user-defined records in the declarative part of any block, subprogram, or package.

A record can be initialized in its declaration:

```
DECLARE
   TYPE TimeTyp IS RECORD ( seconds SMALLINT := 0, minutes SMALLINT := 0,
      hours SMALLINT := 0 );
```

You can use the %TYPE attribute to specify the datatype of a field. You can add the NOT NULL constraint to any field declaration to prevent the assigning of nulls to that field. Fields declared as NOT NULL must be initialized.

```
DECLARE
   TYPE DeptRecTyp IS RECORD
   (
      deptno NUMBER(2) NOT NULL := 99,
      dname  departments.department_name%TYPE,
      loc    departments.location_id%TYPE,
      region regions%ROWTYPE
   );
   dept_rec DeptRecTyp;
BEGIN
   dept_rec.dname := 'PURCHASING';
END;
/
```

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:

```
dept_rec.dname := 'PURCHASING';
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once:

- You can assign one user-defined record to another if they have the same datatype. (Having fields that match exactly is not enough.) You can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes.
- You can use the SELECT or FETCH statement to fetch column values into a record. The columns in the select-list must appear in the same order as the fields in your record.

You can declare and reference nested records. That is, a record can be the component of another record:

```
DECLARE
   TYPE TimeTyp IS RECORD ( minutes SMALLINT, hours SMALLINT );
   TYPE MeetingTyp IS RECORD (
      day     DATE,
      time_of TimeTyp,             -- nested record
      dept    departments%ROWTYPE, -- nested record representing a table row
```

```
      place   VARCHAR2(20),
      purpose VARCHAR2(50) );
   meeting MeetingTyp;
   seminar MeetingTyp;
BEGIN
   seminar.time_of := meeting.time_of;
END;
/
```

You can assign one nested record to another if they have the same datatype:

```
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

The following example defines a `RECORD` type named `DeptRecTyp`, declares a record named `dept_rec`, then selects a row of values into the record:

```
DECLARE
   TYPE DeptRecTyp IS RECORD (
      deptno departments.department_id%TYPE,
      dname  departments.department_name%TYPE,
      loc    departments.location_id%TYPE );
   dept_rec DeptRecTyp;
BEGIN
   SELECT department_id, department_name, location_id INTO dept_rec
      FROM departments WHERE department_id = 20;
END;
/
```

Related Topics

Collections, Functions, Packages, Procedures

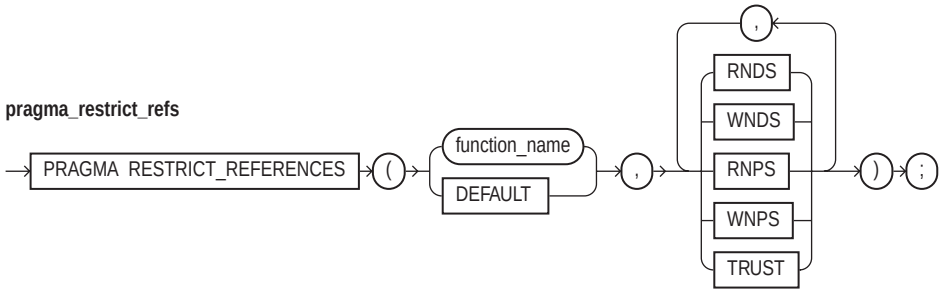
RESTRICT_REFERENCES Pragma

To be callable from SQL statements, a stored function must obey certain "purity" rules, which control side-effects. (See "Controlling Side Effects of PL/SQL Subprograms" on page 8-22.) The fewer side-effects a function has, the better it can be optimized within a query, particular when the `PARALLEL_ENABLE` or `DETERMINISTIC` hints are used. The same rules that apply to the function itself also apply to any functions or procedures that it calls.

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 at compile time, you can use the compiler directive `PRAGMA RESTRICT_REFERENCES`. This pragma asserts that a function does not read and/or write database tables and/or package variables. Functions that do any of these read or write operations are difficult to optimize, because any call might produce different results or encounter errors.

For more information, see *Oracle Database Application Developer's Guide - Fundamentals*.

Syntax



Keyword and Parameter Description

DEFAULT

Specifies that the pragma applies to all subprograms in the package spec or object type spec. You can still declare the pragma for individual subprograms. Such pragmas override the default pragma.

function_name

A user-defined function or procedure.

PRAGMA

Signifies that the statement is a compiler directive. Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they convey information to the compiler.

RNDS

Asserts that the subprogram reads no database state (does not query database tables).

RNPS

Asserts that the subprogram reads no package state (does not reference the values of packaged variables)

TRUST

Asserts that the subprogram can be trusted not to violate one or more rules. This value is needed for functions written in C or Java that are called from PL/SQL, since PL/SQL cannot verify them at run time.

WNDS

Asserts that the subprogram writes no database state (does not modify database tables).

WNPS

Asserts that the subprogram writes no package state (does not change the values of packaged variables).

Usage Notes

You can declare the pragma `RESTRICT_REFERENCES` only in a package spec or object type spec. You can specify up to four constraints (`RNDS`, `RNPS`, `WNDS`, `WNPS`) in any order. To call a function from parallel queries, you must specify all four constraints. No constraint implies another.

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. Skipping these checks can improve performance.

If you specify `DEFAULT` instead of a subprogram name, the pragma applies to all subprograms in the package spec or object type spec (including the system-defined constructor for object types). You can still declare the pragma for individual subprograms, overriding the default pragma.

A `RESTRICT_REFERENCES` pragma can apply to only one subprogram declaration. A pragma that references the name of overloaded subprograms always applies to the most recent subprogram declaration.

Typically, you only specify this pragma for functions. If a function calls procedures, then you need to specify the pragma for those procedures as well.

Examples

This example asserts that the function `BALANCE` writes no database state (`WNDS`) and reads no package state (`RNPS`). That is, it does not issue any DDL or DML statements, and does not refer to any package variables, and neither do any procedures or functions that it calls. It might issue queries or assign values to package variables.

```
CREATE PACKAGE loans AS
   FUNCTION balance(account NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (balance, WNDS, RNPS);
END loans;
/

DROP PACKAGE loans;
```

Related Topics

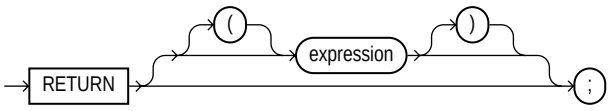
AUTONOMOUS_TRANSACTION Pragma, EXCEPTION_INIT Pragma, SERIALLY_REUSABLE Pragma

RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution 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 8-4.

Syntax

return_statement



Keyword and Parameter Description

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the RETURN statement is executed, the value of `expression` is assigned to the function identifier.

Usage Notes

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. Executing any of them completes the subprogram immediately. The RETURN statement might not be positioned as the last statement in the subprogram.

In procedures, a RETURN statement cannot contain an expression. The statement just returns control to the caller before the normal end of the procedure is reached.

In functions, a RETURN statement *must* contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier. In functions, there must be at least one execution path that leads to a RETURN statement. Otherwise, PL/SQL raises an exception at run time.

The RETURN statement can be used in an anonymous block to exit the block (and all enclosing blocks), but the RETURN statement cannot contain an expression.

Example

The following example demonstrates the RETURN statement using a variable, an expression, or no argument at all:

```
DECLARE
   FUNCTION num_rows (table_name VARCHAR2) RETURN user_tables.num_rows%TYPE
   IS
        howmany user_tables.num_rows%TYPE;
   BEGIN
      EXECUTE IMMEDIATE 'SELECT num_rows FROM user_tables ' ||
    'WHERE table_name = ''' || UPPER(table_name) || ''''
         INTO howmany;
-- A function can compute a value, then return that value.
```

```
      RETURN howmany;
   END num_rows;

   FUNCTION double_it(n NUMBER) RETURN NUMBER
   IS
   BEGIN
-- A function can also return an expression.
      RETURN n * 2;
   END double_it;

   PROCEDURE print_something
   IS
   BEGIN
      dbms_output.put_line('Message 1.');
-- A procedure can end early by issuing RETURN with no value.
      RETURN;
      dbms_output.put_line('Message 2 (never printed).');
   END;
BEGIN
   dbms_output.put_line('EMPLOYEES has ' || num_rows('employees') || ' rows.');
   dbms_output.put_line('Twice 100 is ' || double_it(n => 100) || '.');
   print_something;
END;
/
```

Related Topics

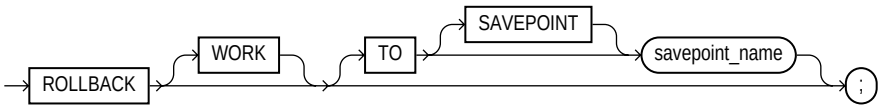
Functions, Procedures

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 6-29.

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

Undoes all database changes (and releases all locks acquired) since the savepoint identified by `savepoint_name` was marked.

SAVEPOINT

Optional, for readability only.

savepoint_name

An undeclared identifier, which marks the current point in the processing of a transaction. For naming conventions, see "Identifiers" on page 2-3.

WORK

Optional, for readability only.

Usage Notes

All savepoints marked after the savepoint to which you roll back are erased. The savepoint to which you roll back is not erased. For example, if you mark savepoints A, B, C, and D in that order, then roll back to savepoint B, only savepoints C and D are erased.

An implicit savepoint is marked before executing an `INSERT`, `UPDATE`, or `DELETE` statement. If the statement fails, a rollback to this implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

In SQL, the `FORCE` clause manually rolls back an in-doubt distributed transaction. PL/SQL does not support this clause. For example, the following statement is not allowed:

```
ROLLBACK WORK FORCE '24.37.85';  -- not allowed
```

In embedded SQL, the RELEASE option frees all Oracle resources (locks and cursors) held by a program and disconnects from the database. PL/SQL does not support this option. For example, the following statement is not allowed:

```
ROLLBACK WORK RELEASE;  -- not allowed
```

Related Topics

COMMIT Statement, SAVEPOINT Statement

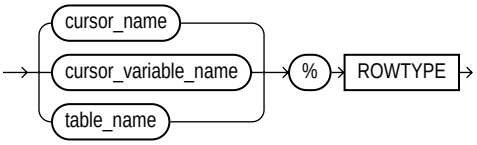
%ROWTYPE Attribute

The %ROWTYPE attribute provides a record type that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. 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 the %ROWTYPE Attribute" on page 2-10.

Syntax

rowtype_attribute



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL strongly typed cursor variable, previously declared within the current scope.

table_name

A database table or view that must be accessible when the declaration is elaborated.

Usage Notes

Declaring variables as the type *table_name*%ROWTYPE is a convenient way to transfer data between database tables and PL/SQL. You create a single variable rather than a separate variable for each column. You do not need to know the name of every column. You refer to the columns using their real names instead of made-up variable names. If columns are later added to or dropped from the table, your code can keep working without changes.

To reference a field in the record, use dot notation (`record_name.field_name`). You can read or write one field at a time this way.

There are two ways to assign values to all fields in a record at once:

- First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.
- You can assign a list of column values to a record by using the SELECT or FETCH statement. The column names must appear in the order in which they were declared. Select-items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases.

Examples

The following example uses %ROWTYPE to declare two records. The first record stores an entire row selected from a table. The second record stores a row fetched from the c1 cursor, which queries a subset of the columns from the table. The example retrieves a single row from the table and stores it in the record, then checks the values of some table columns.

```
DECLARE
   emp_rec   employees%ROWTYPE;
   my_empno  employees.employee_id%TYPE := 100;
   CURSOR c1 IS
      SELECT department_id, department_name, location_id FROM departments;
   dept_rec  c1%ROWTYPE;
BEGIN
   SELECT * INTO emp_rec FROM employees WHERE employee_id = my_empno;
   IF (emp_rec.department_id = 20) AND (emp_rec.salary > 2000) THEN
      NULL;
   END IF;
END;
/
```

Related Topics

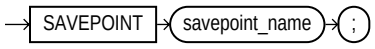
Constants and Variables, Cursors, Cursor Variables, FETCH Statement

SAVEPOINT Statement

The SAVEPOINT statement names and marks the current point in the processing of a transaction. With the ROLLBACK TO statement, savepoints undo parts of a transaction instead of the whole transaction. For more information, see "Overview of Transaction Processing in PL/SQL" on page 6-29.

Syntax

savepoint_statement



Keyword and Parameter Description

savepoint_name

An undeclared identifier, which marks the current point in the processing of a transaction.

Usage Notes

A simple rollback or commit erases all savepoints. When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back remains.

You can reuse savepoint names within a transaction. The savepoint moves from its old position to the current point in the transaction.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent. You can only roll back to the most recently marked savepoint.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction; if the statement raises an unhandled exception, the host environment (such as SQL*Plus) determines what is rolled back.

Related Topics

COMMIT Statement, ROLLBACK Statement

SCN_TO_TIMESTAMP Function

Syntax

```
return_value := SCN_TO_TIMESTAMP(number);
```

Purpose

SCN_TO_TIMESTAMP takes an argument that represents a system change number (SCN) and returns the timestamp associated with that SCN. The returned value has the datatype TIMESTAMP.

Usage Notes

This function is part of the flashback query feature. System change numbers provide a precise way to specify the database state at a moment in time, so that you can see the data as it was at that moment.

Call this function to find out the date and time associated with an SCN that you have stored to use with flashback query.

Examples

```
DECLARE
   right_now TIMESTAMP; yesterday TIMESTAMP; sometime TIMESTAMP;
   scn1 INTEGER; scn2 INTEGER; scn3 INTEGER;
BEGIN
-- Get the current SCN.
   right_now := SYSTIMESTAMP;
   scn1 := TIMESTAMP_TO_SCN(right_now);
   dbms_output.put_line('Current SCN is ' || scn1);

-- Get the SCN from exactly 1 day ago.
   yesterday := right_now - 1;
   scn2 := TIMESTAMP_TO_SCN(yesterday);
   dbms_output.put_line('SCN from yesterday is ' || scn2);

-- Find an arbitrary SCN somewhere between yesterday and today.
-- (In a real program we would have stored the SCN at some significant moment.)
   scn3 := (scn1 + scn2) / 2;
-- Find out what time that SCN was in effect.
   sometime := SCN_TO_TIMESTAMP(scn3);
   dbms_output.put_line('SCN ' || scn3 || ' was in effect at ' ||
TO_CHAR(sometime));
END;
/
```

Related Topics

TIMESTAMP_TO_SCN Function

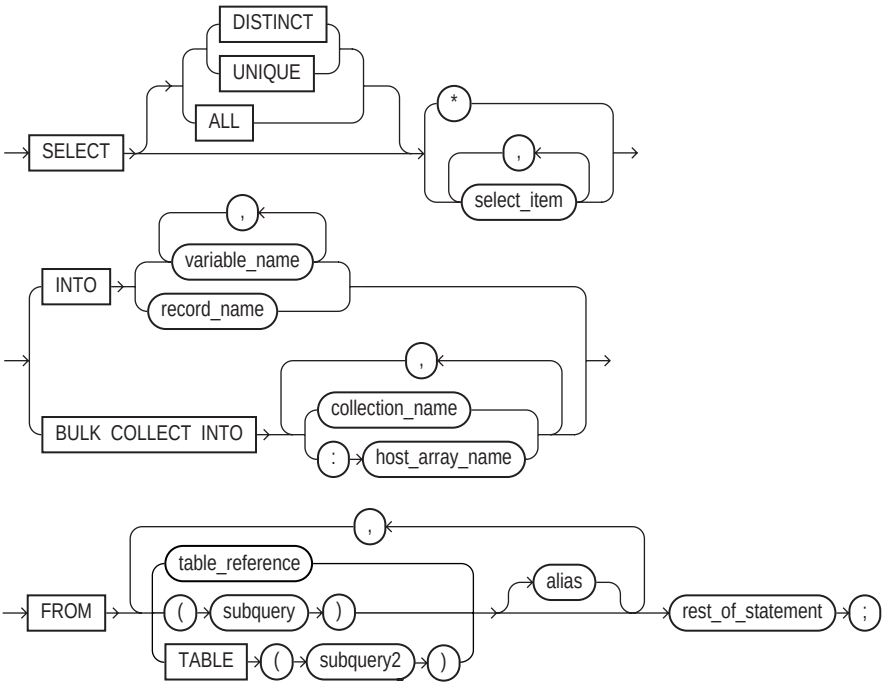
SELECT INTO Statement

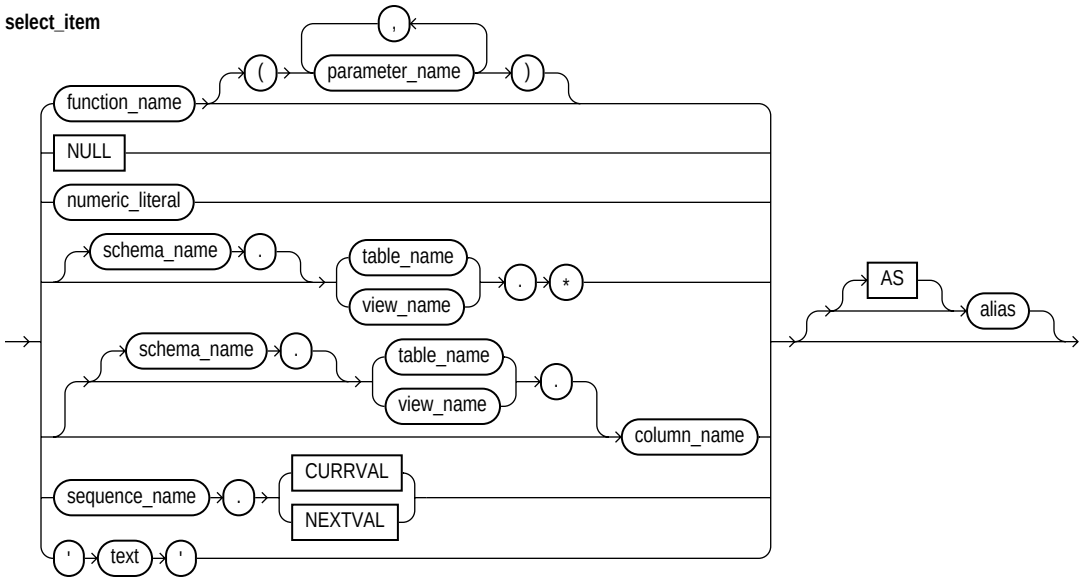
The `SELECT INTO` statement retrieves data from one or more database tables, and assigns the selected values to variables or collections. For a full description of the `SELECT` statement, see *Oracle Database SQL Reference*.

In its default usage (`SELECT ... INTO`), this statement retrieves one or more columns from a single row. In its bulk usage (`SELECT ... BULK COLLECT INTO`), this statement retrieves an entire result set at once.

Syntax

select_into_statement





Keyword and Parameter Description

alias

Another (usually short) name for the referenced column, table, or view.

BULK COLLECT

Stores result values in one or more collections, for faster queries than loops with `FETCH` statements. For more information, see "Reducing Loop Overhead for DML Statements and Queries (FORALL, BULK COLLECT)" on page 11-7.

collection_name

A declared collection into which `select_item` values are fetched. For each `select_item`, there must be a corresponding, type-compatible collection in the list.

function_name

A user-defined function.

host_array_name

An array (declared in a PL/SQL host environment and passed to PL/SQL as a bind variable) into which `select_item` values are fetched. For each `select_item`, there must be a corresponding, type-compatible array in the list. Host arrays must be prefixed with a colon.

numeric_literal

A literal that represents a number or a value implicitly convertible to a number.

parameter_name

A formal parameter of a user-defined function.

record_name

A user-defined or %ROWTYPE record into which rows of values are fetched. For each select_item value returned by the query, there must be a corresponding, type-compatible field in the record.

rest_of_statement

Anything that can follow the FROM clause in a SQL SELECT statement (except the SAMPLE clause).

schema_name

The schema containing the table or view. If you omit schema_name, Oracle assumes the table or view is in your schema.

subquery

A SELECT statement that provides a set of rows for processing. Its syntax is like that of select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 13-123.

table_reference

A table or view that must be accessible when you execute the SELECT statement, and for which you must have SELECT privileges. For the syntax of table_reference, see "DELETE Statement" on page 13-41.

TABLE (subquery2)

The operand of TABLE is a SELECT statement that returns a single column value, which must be a nested table or a varray. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

variable_name

A previously declared variable into which a select_item value is fetched. For each select_item value returned by the query, there must be a corresponding, type-compatible variable in the list.

Usage Notes

By default, a SELECT INTO statement must return only one row. Otherwise, PL/SQL raises the predefined exception TOO_MANY_ROWS and the values of the variables in the INTO clause are undefined. Make sure your WHERE clause is specific enough to only match one row

If no rows are returned, PL/SQL raises NO_DATA_FOUND. You can guard against this exception by selecting the result of an aggregate function, such as COUNT(*) or AVG(), where practical. These functions are guaranteed to return a single value, even if no rows match the condition.

A SELECT ... BULK COLLECT INTO statement can return multiple rows. You must set up collection variables to hold the results. You can declare associative arrays or nested tables that grow as needed to hold the entire result set.

The implicit cursor SQL and its attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN provide information about the execution of a SELECT INTO statement.

Examples

The following example demonstrates using the `SELECT INTO` statement to query a single value into a PL/SQL variable, entire columns into PL/SQL collections, or entire rows into a PL/SQL collection of records:

```
DECLARE
   howmany NUMBER;
   some_first employees.first_name%TYPE;
   some_last employees.last_name%TYPE;
   some_employee employees%ROWTYPE;
   TYPE first_typ IS TABLE OF employees.first_name%TYPE INDEX BY PLS_INTEGER;
   TYPE last_typ IS TABLE OF employees.first_name%TYPE INDEX BY PLS_INTEGER;
   first_names first_typ;
   last_names last_typ;
   CURSOR c1 IS SELECT first_name, last_name FROM employees;
   TYPE name_typ IS TABLE OF c1%ROWTYPE INDEX BY PLS_INTEGER;
   all_names name_typ;
   TYPE emp_typ IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
   all_employees emp_typ;
BEGIN
-- Query a single value and store it in a variable.
   SELECT COUNT(*) INTO howmany FROM user_tables;
   dbms_output.put_line('This schema owns ' || howmany || ' tables.');

-- Query multiple columns from one row, and store them in variables.
   SELECT first_name, last_name INTO some_first, some_last
      FROM employees WHERE ROWNUM < 2;
   dbms_output.put_line('Random employee: ' || some_first ||
      ' ' || some_last);

-- Query a single row and store it in a record.
   SELECT * INTO some_employee FROM employees WHERE ROWNUM < 2;

-- Query multiple columns from multiple rows, and store them in a collection
-- of records.
   SELECT first_name, last_name BULK COLLECT INTO all_names FROM EMPLOYEES;

-- Query multiple columns from multiple rows, and store them in separate
-- collections. (Generally less useful than a single collection of records.)
   SELECT first_name, last_name
      BULK COLLECT INTO first_names, last_names
      FROM EMPLOYEES;

-- Query an entire (small!) table and store the rows
-- in a collection of records. Now you can manipulate the data
-- in-memory without any more I/O.
   SELECT * BULK COLLECT INTO all_employees FROM employees;
END;
/
```

Related Topics

Assignment Statement, FETCH Statement, %ROWTYPE Attribute

SERIALLY_REUSABLE Pragma

The pragma SERIALLY_REUSABLE indicates that the package state is needed only for the duration of one call to the server (for example, a PL/SQL anonymous block, an OCI call to the database or a stored procedure call through a database link). After this call, the storage for the package variables can be reused, reducing the memory overhead for long-running sessions. For more information, see *Oracle Database Application Developer's Guide - Fundamentals*.

Syntax

serially_reusable_pragma

Keyword and Parameter Description

PRAGMA

Signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler.

Usage Notes

This pragma is appropriate for packages that declare large temporary work areas that are used once and not needed during subsequent database calls in the same session.

You can mark a bodiless package as serially reusable. If a package has a spec and body, you must mark both. You cannot mark only the body.

The global memory for serially reusable packages is pooled in the System Global Area (SGA), not allocated to individual users in the User Global Area (UGA). That way, the package work area can be reused. When the call to the server ends, the memory is returned to the pool. Each time the package is reused, its public variables are initialized to their default values or to NULL.

Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are called from SQL statements. If you try, Oracle generates an error.

Examples

The following example creates 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;
/

DROP PACKAGE pkg1;
```

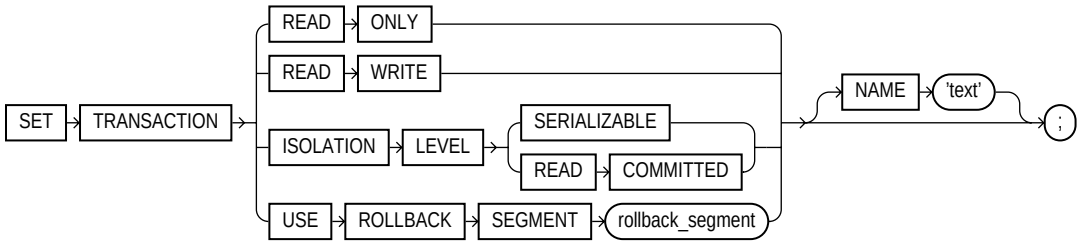
Related Topics

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 6-32.

Syntax



Keyword and Parameter Description

READ ONLY

Establishes the current transaction as read-only, so that subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

READ WRITE

Establishes the current transaction as read-write. The use of READ WRITE does not affect other users or transactions. If the transaction executes a data manipulation statement, Oracle assigns the transaction to a rollback segment.

ISOLATION LEVEL

Specifies how to handle transactions that modify the database.

SERIALIZABLE: If a serializable transaction tries to execute a SQL data manipulation statement that modifies any table already modified by an uncommitted transaction, the statement fails.

To enable SERIALIZABLE mode, your DBA must set the Oracle initialization parameter COMPATIBLE to 7.3.0 or higher.

READ COMMITTED: If a transaction includes SQL data manipulation statements that require row locks held by another transaction, the statement waits until the row locks are released.

USE ROLLBACK SEGMENT

Assigns the current transaction to the specified rollback segment and establishes the transaction as read-write. You cannot use this parameter with the READ ONLY parameter in the same transaction because read-only transactions do not generate rollback information.

NAME

Specifies a name or comment text for the transaction. This is better than using the `COMMIT COMMENT` feature because the name is available while the transaction is running, making it easier to monitor long-running and in-doubt transactions.

Usage Notes

The `SET TRANSACTION` statement must be the first SQL statement in the transaction and can appear only once in the transaction.

Example

The following example establishes a read-only transaction:

```
BEGIN
   COMMIT;  -- end previous transaction
   SET TRANSACTION READ ONLY;
   FOR person IN (SELECT last_name FROM employees WHERE ROWNUM < 10)
   LOOP
      dbms_output.put_line(person.last_name);
   END LOOP;
   dbms_output.put_line('------------------');
   FOR dept IN (SELECT department_name FROM departments WHERE ROWNUM < 10)
   LOOP
      dbms_output.put_line(dept.department_name);
   END LOOP;
   COMMIT;  -- end read-only transaction
END;
/
```

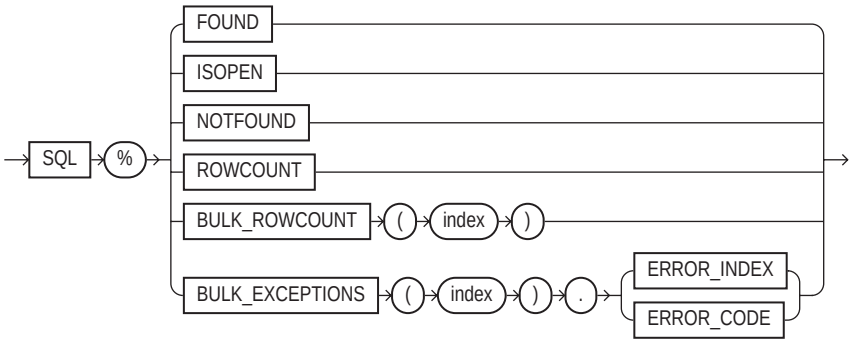
Related Topics

COMMIT Statement, ROLLBACK Statement, SAVEPOINT Statement

SQL Cursor

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicit cursor. In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. They provide information about the execution of data manipulation statements. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. For more information, see "Querying Data with PL/SQL" on page 6-9.

Syntax



Keyword and Parameter Description

%BULK_ROWCOUNT

A composite attribute designed for use with the FORALL statement. This attribute acts like an index-by table. Its *i*th element stores the number of rows processed by the *i*th execution of an UPDATE or DELETE statement. If the *i*th execution affects no rows, %BULK_ROWCOUNT(i) returns zero.

%BULK_EXCEPTIONS

An associative array that stores information about any exceptions encountered by a FORALL statement that uses the SAVE EXCEPTIONS clause. You must loop through its elements to determine where the exceptions occurred and what they were. For each index value i between 1 and SQL%BULK_EXCEPTIONS.COUNT, SQL%BULK_EXCEPTIONS(i).ERROR_INDEX specifies which iteration of the FORALL loop caused an exception. SQL%BULK_EXCEPTIONS(i).ERROR_CODE specifies the Oracle error code that corresponds to the exception.

%FOUND

Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.

%ISOPEN

Always returns FALSE, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%NOTFOUND

The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

%ROWCOUNT

Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

SQL

The name of the Oracle implicit cursor.

Usage Notes

You can use cursor attributes in procedural statements but not in SQL statements. Before Oracle opens the SQL cursor automatically, the implicit cursor attributes return NULL.

The values of cursor attributes always refer to the most recently executed SQL statement, wherever that statement appears. It might be in a different scope. If you want to save an attribute value for later use, assign it to a 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. 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 returns FALSE.

%BULK_ROWCOUNT is *not* maintained for bulk inserts; that would be redundant. For example, the following FORALL statement inserts one row per iteration. After each iteration, %BULK_ROWCOUNT returns 1:

```
CREATE TABLE num_table (n NUMBER);

DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   nums NumList := NumList(1,3,5,7,11,13,17);
BEGIN
   FORALL i IN nums.FIRST .. nums.LAST
      INSERT INTO num_table (n) VALUES (nums(i));

   FOR i IN nums.FIRST .. nums.LAST
   LOOP
      dbms_output.put_line('Inserted ' || SQL%BULK_ROWCOUNT(i) || ' row(s)'
         || ' on iteration ' || i);
   END LOOP;
END;
/

DROP TABLE num_table;
```

You can use the scalar attributes %FOUND, %NOTFOUND, and %ROWCOUNT with bulk binds. For example, %ROWCOUNT returns the total number of rows processed by all executions of the SQL statement.

Although %FOUND and %NOTFOUND refer only to the last execution of the SQL statement, you can use %BULK_ROWCOUNT to infer their values for individual

executions. For example, when %BULK_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

Examples

The following example inserts a new row only if an update affects no rows:

```
CREATE TABLE visitors (email VARCHAR2(128), pages_visited INTEGER DEFAULT 1);

CREATE OR REPLACE PROCEDURE someone_visited (visitor_email visitors.email%TYPE)
AS
BEGIN
   UPDATE visitors SET pages_visited = pages_visited + 1
      WHERE email = visitor_email;
   IF SQL%NOTFOUND THEN
      INSERT INTO visitors (email) VALUES (visitor_email);
      dbms_output.put_line('Adding ' || visitor_email || ' to the table.');
   ELSE
      dbms_output.put_line('Incremented counter for ' || visitor_email || '.');
   END IF;
END;
/

DECLARE
   visitor_email visitors.email%TYPE := 'fred@fictional_domain.com';
BEGIN
   someone_visited(visitor_email);
   someone_visited(visitor_email);
END;
/

DROP TABLE visitors;
DROP PROCEDURE someone_visited;
```

The following example raises an exception if more than 10 rows are deleted:

```
CREATE TABLE temp AS SELECT object_name name FROM user_objects;

DECLARE
   large_deletion EXCEPTION;
   rows_deleted NUMBER;
BEGIN
   DELETE FROM temp WHERE name LIKE '%A%';
   rows_deleted := SQL%ROWCOUNT;
   COMMIT;
   IF rows_deleted > 10 THEN
       RAISE large_deletion;
   END IF;

   dbms_output.put_line('Nothing unusual detected.');

   EXCEPTION
      WHEN large_deletion THEN
         dbms_output.put_line('Recording deletion of ' ||
            rows_deleted || ' rows in case of error.');
END;
/

DROP TABLE temp;
```

The following example uses %BULK_ROWCOUNT. After the FORALL statement completes, the program checks how many rows were updated by the third UPDATE:

```
CREATE TABLE num_table (n NUMBER);

DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   nums NumList := NumList(1,3,5,5,11,5,5);
BEGIN
   FORALL i IN nums.FIRST .. nums.LAST
      INSERT INTO num_table (n) VALUES (nums(i));

-- All the numbers in the table will be squared.
-- Some updates will affect more rows than others.
   FORALL j IN nums.FIRST .. nums.LAST
      UPDATE num_table SET n = n * n
            WHERE n = nums(j);

   FOR k IN nums.FIRST .. nums.LAST
   LOOP
      dbms_output.put_line('Update #' || k || ' affected ' ||
         SQL%BULK_ROWCOUNT(k) || ' rows.');
   END LOOP;
END;
/

DROP TABLE num_table;
```

Related Topics

Cursors, Cursor Attributes, FORALL Statement, "Handling FORALL Exceptions with the %BULK_EXCEPTIONS Attribute" on page 11-13

SQLCODE Function

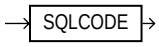
The function SQLCODE returns the number code of the most recent exception.

For internal exceptions, SQLCODE returns the number of the associated Oracle error. The number that SQLCODE returns is negative unless the Oracle error is *no data found*, in which case SQLCODE returns +100.

For user-defined exceptions, SQLCODE returns +1, or a value you assign if the exception is associated with an Oracle error number through pragma EXCEPTION_INIT.

Syntax

sqlcode_function



Usage Notes

SQLCODE is only useful in an exception handler. Outside a handler, SQLCODE always returns 0. SQLCODE is especially useful in the OTHERS exception handler, because it lets you identify which internal exception was raised.

You cannot use SQLCODE directly in a SQL statement. Assign the value of SQLCODE to a local variable first.

When using pragma RESTRICT_REFERENCES to assert the purity of a stored function, you cannot specify the constraints WNPS and RNPS if the function calls SQLCODE.

Example

The following example inserts the value of SQLCODE into an audit table:

```
CREATE TABLE errors (code NUMBER, message VARCHAR2(128), happened TIMESTAMP);
DECLARE
   name employees.last_name%TYPE;
   my_code NUMBER;
   my_errm VARCHAR2(32000);
BEGIN
   SELECT last_name INTO name FROM employees WHERE employee_id = -1;
   EXCEPTION
      WHEN OTHERS THEN
         my_code := SQLCODE;
         my_errm := SQLERRM;
         dbms_output.put_line('Error code ' || my_code || ': ' || my_errm);
-- Normally we would call another procedure, declared with PRAGMA
-- AUTONOMOUS_TRANSACTION, to insert information about errors.
         INSERT INTO errors VALUES (my_code, my_errm, SYSTIMESTAMP);
END;
/
DROP TABLE errors;
```

Related Topics

Exceptions, SQLERRM Function, "Retrieving the Error Code and Error Message: SQLCODE and SQLERRM" on page 10-14.

SQLERRM Function

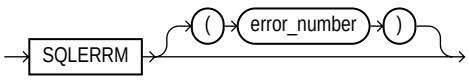
The function SQLERRM returns the error message associated with its error-number argument. If the argument is omitted, it returns the error message associated with the current value of SQLCODE. SQLERRM with no argument is useful only in an exception handler. Outside a handler, SQLERRM with no argument always returns the 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 10-14.

Syntax

sqlerrm_function



Keyword and Parameter Description

error_number

A valid Oracle error number. For a list of Oracle errors (ones prefixed by ORA-), see **Oracle Database Error Messages**.

Usage Notes

SQLERRM is especially useful in the OTHERS exception handler, where it lets you identify which internal exception was raised.

The error number passed to SQLERRM 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. Assign the value of SQLERRM to a local variable first:

```
my_sqlerrm := SQLERRM;
...
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

The following example retrieves the error message associated with an unhandled exception, and stores it in an audit table. The SUBSTR function truncates the message if it is too long to fit in the table.

```
CREATE TABLE errors (code NUMBER, message VARCHAR2(128), happened TIMESTAMP);
DECLARE
   name employees.last_name%TYPE;
   my_code NUMBER;
   my_errm VARCHAR2(32000);
BEGIN
   SELECT last_name INTO name FROM employees WHERE employee_id = -1;
   EXCEPTION
      WHEN OTHERS THEN
         my_code := SQLCODE;
         my_errm := SQLERRM;
         dbms_output.put_line('Error code ' || my_code || ': ' || my_errm);
-- Normally we would call another procedure, declared with PRAGMA
-- AUTONOMOUS_TRANSACTION, to insert information about errors.
         INSERT INTO errors VALUES (my_code, my_errm, SYSTIMESTAMP);
END;
/
DROP TABLE errors;
```

Related Topics

Exceptions, SQLCODE Function

TIMESTAMP_TO_SCN Function

Syntax

```
return_value := TIMESTAMP_TO_SCN(timestamp);
```

Purpose

TIMESTAMP_TO_SCN takes an argument that represents a precise time, and returns the system change number (SCN) of the database at that moment in time. The returned value has the datatype NUMBER.

Usage Notes

This function is part of the flashback query feature. System change numbers provide a precise way to specify the database state at a moment in time, so that you can see the data as it was at that moment.

Call this function to find out the system change number associated with the date and time to which you want to "flash back".

Examples

```
DECLARE
   right_now TIMESTAMP; yesterday TIMESTAMP; sometime TIMESTAMP;
   scn1 INTEGER; scn2 INTEGER; scn3 INTEGER;
BEGIN
-- Get the current SCN.
   right_now := SYSTIMESTAMP;
   scn1 := TIMESTAMP_TO_SCN(right_now);
   dbms_output.put_line('Current SCN is ' || scn1);

-- Get the SCN from exactly 1 day ago.
   yesterday := right_now - 1;
   scn2 := TIMESTAMP_TO_SCN(yesterday);
   dbms_output.put_line('SCN from yesterday is ' || scn2);

-- Find an arbitrary SCN somewhere between yesterday and today.
-- (In a real program we would have stored the SCN at some significant moment.)
   scn3 := (scn1 + scn2) / 2;
-- Find out what time that SCN was in effect.
   sometime := SCN_TO_TIMESTAMP(scn3);
   dbms_output.put_line('SCN ' || scn3 || ' was in effect at ' ||
TO_CHAR(sometime));
END;
/
```

Related Topics

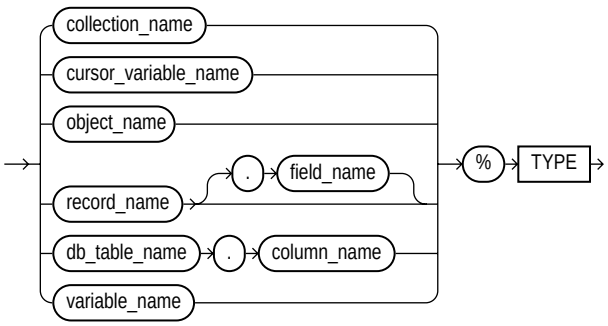
SCN_TO_TIMESTAMP Function

%TYPE Attribute

The %TYPE attribute lets use the datatype of a field, record, nested table, database column, or variable in your own declarations, instead of hardcoding the type names. You can use the %TYPE attribute as a datatype specifier when declaring constants, variables, fields, and parameters. If the types that you reference change, your declarations are automatically updated. This technique saves you from making code changes when, for example, the length of a VARCHAR2 column is increased. For more information, see "Using the %TYPE Attribute" on page 2-9.

Syntax

type_attribute



Keyword and Parameter Description

collection_name

A nested table, index-by table, or varray previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

db_table_name.column_name

A table and column that must be accessible when the declaration is elaborated.

object_name

An instance of an object type, previously declared within the current scope.

record_name

A user-defined or %ROWTYPE record, previously declared within the current scope.

record_name.field_name

A field in a user-defined or %ROWTYPE record, previously declared within the current scope.

variable_name

A variable, previously declared in the same scope.

Usage Notes

The %TYPE attribute is particularly useful when declaring variables, fields, and parameters that refer to database columns. Your code can keep working even when the lengths or types of the columns change.

The NOT NULL column constraint is not inherited by items declared using %TYPE.

Examples

```
DECLARE
-- We know that BUFFER2 and BUFFER3 will be big enough to hold
-- the answers. If we have to increase the size of BUFFER1, the
-- other variables will change size as well.
   buffer1 VARCHAR2(13) := 'abCdefGhiJklm';
   buffer2 buffer1%TYPE := UPPER(buffer1);
   buffer3 buffer1%TYPE := LOWER(buffer1);

-- We know that this variable will be able to hold the contents
-- of this table column. If the table is altered to make the
-- column longer or shorter, this variable will change size as well.
   tname user_tables.table_name%TYPE;

-- %TYPE is great for subprogram parameters too, no need to
-- recompile the subprogram if the table column changes.
   PROCEDURE print_table_name(the_name user_tables.table_name%TYPE)
   IS
   BEGIN
      dbms_output.put_line('Table = ' || the_name);
   END;
BEGIN
   SELECT table_name INTO tname FROM user_tables WHERE ROWNUM < 2;
   print_table_name(tname);
END;
/
```

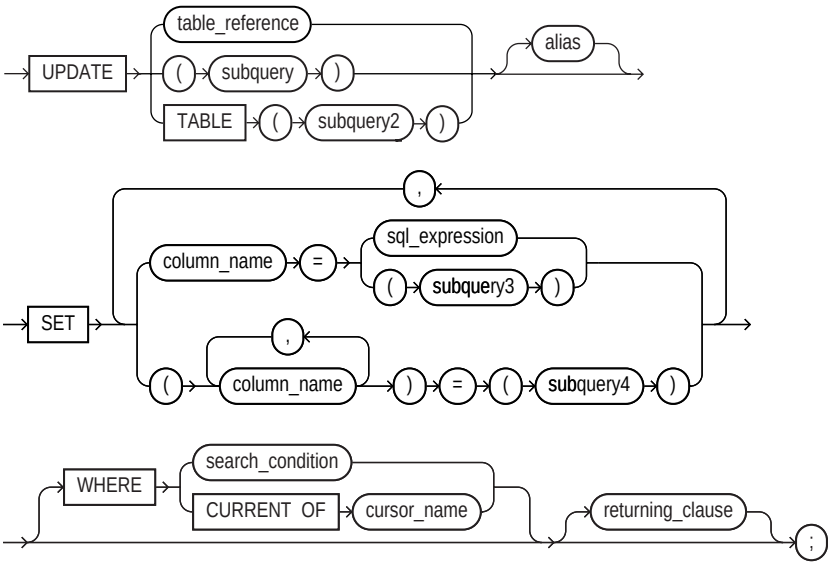
Constants and Variables, %ROWTYPE Attribute

UPDATE Statement

The UPDATE statement changes the values of specified columns in one or more rows in a table or view. For a full description of the UPDATE statement, see *Oracle Database SQL Reference*.

Syntax

update_statement



Keyword and Parameter Description

alias

Another (usually short) name for the referenced table or view, typically used in the WHERE clause.

column_name

The column (or one of the columns) to be updated. It must be the name of a column in the referenced table or view. A column name cannot be repeated in the column_name list. Column names need not appear in the UPDATE statement in the same order that they appear in the table or view.

returning_clause

Returns values from updated rows, eliminating the need to SELECT the rows afterward. You can retrieve the column values into variables or host variables, or into collections or host arrays. You cannot use the RETURNING clause for remote or parallel updates. If the statement does not affect any rows, the values of the variables specified in the RETURNING clause are undefined. For the syntax of returning_clause, see "DELETE Statement" on page 13-41.

SET column_name = sql_expression

This clause assigns the value of sql_expression to the column identified by column_name. If sql_expression contains references to columns in the table being

updated, the references are resolved in the context of the current row. The old column values are used on the right side of the equal sign.

SET column_name = (subquery3)

Assigns the value retrieved from the database by `subquery3` to the column identified by `column_name`. The subquery must return exactly one row and one column.

SET (column_name, column_name, ...) = (subquery4)

Assigns the values retrieved from the database by `subquery4` to the columns in the `column_name` list. The subquery must return exactly one row that includes all the columns listed.

The column values returned by the subquery are assigned to the columns in the column list in order. The first value is assigned to the first column in the list, the second value is assigned to the second column in the list, and so on.

The following example creates a table with correct employee IDs but garbled names. Then it runs an `UPDATE` statement with a correlated query, to retrieve the correct names from the `EMPLOYEES` table and fix the names in the new table.

```
-- Create a table with all the right IDs, but messed-up names.
CREATE TABLE e1 AS
   SELECT employee_id,
          UPPER(first_name) first_name,
          TRANSLATE(last_name,'aeiou','12345') last_name
   FROM employees;

BEGIN
-- Display the first 5 names to show they're messed up.
   FOR person IN (SELECT * FROM e1 WHERE ROWNUM < 6)
   LOOP
      dbms_output.put_line(person.first_name || ' ' || person.last_name);
   END LOOP;
   UPDATE e1 SET (first_name, last_name) =
      (SELECT first_name, last_name FROM employees
         WHERE employee_id = e1.employee_id);
   dbms_output.put_line('*** Updated ' || SQL%ROWCOUNT || ' rows. ***');
-- Display the first 5 names to show they've been fixed up.
   FOR person IN (SELECT * FROM e1 WHERE ROWNUM < 6)
   LOOP
      dbms_output.put_line(person.first_name || ' ' || person.last_name);
   END LOOP;
END;
/

DROP TABLE e1;
```

sql_expression

Any valid SQL expression. For more information, see *Oracle Database SQL Reference*.

subquery

A `SELECT` statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 13-123.

table_reference

A table or view that must be accessible when you execute the UPDATE statement, and for which you must have UPDATE privileges. For the syntax of table_reference, see "DELETE Statement" on page 13-41.

TABLE (subquery2)

The operand of TABLE is a SELECT statement that returns a single column value, which must be a nested table or a varray. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

WHERE CURRENT OF cursor_name

Refers to the latest row processed by the FETCH statement associated with the specified cursor. The cursor must be FOR UPDATE and must be open and positioned on a row.

If the cursor is not open, the CURRENT OF clause causes an error. If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception NO_DATA_FOUND.

WHERE search_condition

Chooses which rows to update in the database table. Only rows that meet the search condition are updated. If you omit this clause, all rows in the table are updated.

Usage Notes

You can use the UPDATE WHERE CURRENT OF statement after a fetch from an open cursor (including fetches done by a cursor FOR loop), provided the associated query is FOR UPDATE. This statement updates the row that was just fetched.

The implicit cursor SQL and the cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of an UPDATE statement.

Examples

The following example demonstrates how to update table rows based on conditions, and how to store the updated values, columns, or entire rows in PL/SQL variables:

```
-- Create some rows with values in all caps like (EMPLOYEES,TABLE)
-- and (EMP_JOB_IX,INDEX).
CREATE TABLE my_objects AS SELECT object_name, object_type FROM user_objects;

DECLARE
   my_name my_objects.object_name%TYPE;
   my_type my_objects.object_type%TYPE;
   TYPE name_typ IS TABLE OF my_objects.object_name%TYPE INDEX BY PLS_INTEGER;
   TYPE type_typ IS TABLE OF my_objects.object_type%TYPE INDEX BY PLS_INTEGER;
   all_names name_typ;
   all_types type_typ;
   TYPE table_typ IS TABLE OF my_objects%ROWTYPE INDEX BY PLS_INTEGER;
   all_rows table_typ;
BEGIN
-- Show the first 10 rows as they originally were.
   FOR obj IN (SELECT * FROM my_objects WHERE ROWNUM < 11)
   LOOP
      dbms_output.put_line('Name = ' || obj.object_name || ', type = ' ||
         obj.object_type);
   END LOOP;
```

```
   UPDATE my_objects SET object_name = LOWER(object_name)
       WHERE object_type = 'TABLE';
   dbms_output.put_line('*** Updated ' || SQL%ROWCOUNT || ' rows. ***');
-- Show the first 10 rows after the update.
-- Only some of the names (the table names) have been changed to lowercase.
   FOR obj IN (SELECT * FROM my_objects WHERE ROWNUM < 11)
   LOOP
      dbms_output.put_line('Name = ' || obj.object_name || ', type = ' ||
         obj.object_type);
   END LOOP;

-- Update a single row, and store the values of updated (or unchanged)
-- columns in variables.
   UPDATE my_objects SET object_name = INITCAP(object_name)
      WHERE object_name = 'employees'
      RETURNING object_name, object_type INTO my_name, my_type;

   dbms_output.put_line('*** Updated ' || SQL%ROWCOUNT || ' rows. ***');
   dbms_output.put_line('Affected this row: ' || my_name || ', ' || my_type);

-- Update many rows, storing the values of updated (or unchanged)
-- columns in collections of records. Can't use 'RETURNING *', have
-- to list the columns individually.
   UPDATE my_objects SET object_name = INITCAP(object_name)
      WHERE object_type IN ('TRIGGER','VIEW','SEQUENCE')
      RETURNING object_name, object_type BULK COLLECT INTO all_rows;

   dbms_output.put_line('*** Updated ' || SQL%ROWCOUNT || ' rows. ***');
   FOR i IN all_rows.FIRST .. all_rows.LAST
   LOOP
      dbms_output.put_line('Affected this row: ' || all_rows(i).object_name || ',
' || all_rows(i).object_type);
   END LOOP;

-- Update many rows, storing the values of updated (or unchanged)
-- columns in separate collections. (Generally less useful than using
-- collections of records as above.)
   UPDATE my_objects SET object_name = INITCAP(object_name)
      WHERE object_type IN ('INDEX','PROCEDURE')
      RETURNING object_name, object_type BULK COLLECT INTO all_names, all_types;

   dbms_output.put_line('*** Updated ' || SQL%ROWCOUNT || ' rows. ***');
   FOR i IN all_names.FIRST .. all_names.LAST
   LOOP
      dbms_output.put_line('Affected this row: ' || all_names(i) || ', ' ||
all_types(i));
   END LOOP;

END;
/

DROP TABLE my_objects;
```

Related Topics

DELETE Statement, FETCH Statement, INSERT Statement

A

Sample PL/SQL Programs

This appendix tells you where to find collections of sample PL/SQL programs, for your own study and testing.

Where to Find PL/SQL Sample Programs

You can find some sample programs in the PL/SQL demo directory. For the location of the directory, see the Oracle installation guide for your system. These samples are typically older ones based on the `SCOTT` schema, with its `EMP` and `DEPT` tables.

Most examples in this book have been made into complete programs that you can run under the `HR` sample schema, with its `EMPLOYEES` and `DEPARTMENTS` tables.

The Oracle Technology Network web site has a PL/SQL section with many sample programs to download, at `http://otn.oracle.com/tech/pl_sql/`. These programs demonstrate many language features, particularly the most recent ones. You can use some of the programs to compare performance of PL/SQL across database releases.

For examples of calling PL/SQL from other languages, see *Oracle Database Java Developer's Guide* and *Pro*C/C++ Programmer's Guide*.

Exercises for the Reader

Here are some PL/SQL programming constructs that are helpful to know. After learning from the sample programs in this book and on the web, check to see that you are familiar with writing each of these constructs.

- An anonymous PL/SQL block.
- A PL/SQL stored procedure.
- A SQL `CALL` statement that invokes a stored procedure. An anonymous block that invokes the stored procedure.
- A PL/SQL stored function.
- A SQL query that calls the stored function.
- A PL/SQL package.
- An anonymous block or a stored procedure that calls a packaged procedure. A SQL query that calls a packaged function.
- A SQL*Plus script, or a set of scripts called from a master script, that creates a set of procedures, functions, and packages.

- A FORALL statement (instead of a regular loop) to issue multiple INSERT, UPDATE, or DELETE statements.

B

Understanding CHAR and VARCHAR2 Semantics in PL/SQL

This appendix explains the semantic differences between the CHAR and VARCHAR2 base types. These subtle but important differences come into play when you assign, compare, insert, update, select, or fetch character values.

This appendix contains these topics:

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. Information about trailing blanks in the original value is lost. In the following example, the value assigned to `last_name` includes six trailing blanks, not just one:

```
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

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

the following assignment raises VALUE_ERROR:

```
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

```
last_name1 VARCHAR2(10) := 'COLES';
last_name2 VARCHAR2(10) := 'COLEMAN';
```

the following IF condition is true:

```
IF last_name1 > last_name2 THEN ...
```

The SQL standard requires that two character values being compared have equal lengths. If both values in a comparison have datatype CHAR, *blank-padding* semantics are used: 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

```
last_name1 CHAR(5) := 'BELLO';
last_name2 CHAR(10) := 'BELLO   ';  -- note trailing blanks
```

the following IF condition is true:

```
IF last_name1 = last_name2 THEN ...
```

If either value in a comparison has datatype VARCHAR2, *non-blank-padding* semantics are used: when comparing character values of unequal length, PL/SQL makes no adjustments and uses the exact lengths. For example, given the declarations

```
last_name1 VARCHAR2(10) := 'DOW';
last_name2 VARCHAR2(10) := 'DOW   ';  -- note trailing blanks
```

the following IF condition is false:

```
IF last_name1 = last_name2 THEN ...
```

If a VARCHAR2 value is compared to a CHAR value, non-blank-padding semantics are used. But, remember, 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. Given the declarations

```
last_name1 VARCHAR2(10) := 'STAUB';
last_name2 CHAR(10)     := 'STAUB';  -- PL/SQL blank-pads value
```

the following IF condition is false because the value of last_name2 includes five trailing blanks:

```
IF last_name1 = last_name2 THEN ...
```

All string literals have datatype CHAR. 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 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:

```
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.

C

Obfuscating Source Code with the PL/SQL Wrap Utility

This appendix shows you how to run the `wrap` utility. `wrap` is a standalone program that obfuscates PL/SQL source code, so that you can deliver PL/SQL applications without exposing your source code.

This appendix contains these topics:

Advantages of Wrapping PL/SQL Procedures

- By hiding application internals, the wrap utility makes it difficult for other developers to misuse your application, or business competitors to see your algorithms.
- Your code is not visible through the `USER_SOURCE`, `ALL_SOURCE`, or `DBA_SOURCE` data dictionary views.
- SQL*Plus can process wrapped files. You can obfuscate source files that create PL/SQL procedures and packages.
- The Import and Export utilities accept wrapped files. You can back up or move wrapped procedures.

Running the PL/SQL 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]
```

Note: Do not use any spaces around the equal signs.

`input_file` is the name of a file containing SQL statements, that you typically run using SQL*Plus. If you omit the file extension, an extension of `.sql` is assumed. For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql
```

You can also specify a different file extension:

```
wrap iname=/mydir/myfile.src
```

output_file is the name of the obfuscated file that is created. The oname option is optional, because the output file name defaults to that of the input file and its extension defaults to .plb. For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

You can use the option oname to specify a different file name and extension:

```
wrap iname=/mydir/myfile oname=/yourdir/yourfile.out
```

Input and Output Files for the PL/SQL Wrap Utility

The input file can contain any combination of SQL statements. Most statements are passed through unchanged. CREATE statements that define subprograms, packages, or object types are obfuscated; their bodies are replaced by a scrambled form that the PL/SQL compiler understands.

The following CREATE statements are obfuscated:

```
CREATE [OR REPLACE] FUNCTION function_name
CREATE [OR REPLACE] PROCEDURE procedure_name
CREATE [OR REPLACE] PACKAGE package_name
CREATE [OR REPLACE] PACKAGE BODY package_name
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```

Note: The CREATE [OR REPLACE] TRIGGER statement, and BEGIN..END anonymous blocks, are *not* obfuscated.

All other SQL statements are passed unchanged to the output file. Most comment lines are deleted. C-style comments (delimited by /* */) are preserved when they occur in the middle of a SQL statement. Comments are also preserved when they occur immediately after the CREATE statement, before the obfuscated body starts.

The output file is a text file, which you can run in SQL*Plus to set up your PL/SQL procedures, functions, and packages:

```
SQL> @wrapped_file_name.plb;
```

Tips:

- 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.
- PL/SQL source inside wrapped files cannot be edited. To change wrapped PL/SQL code, edit the original source file and wrap it again. You can either hold off on wrapping your code until it is ready for shipment to end-users, or include the wrapping operation as part of your build environment.
- To be sure that all the important parts of your source code are obfuscated, view the wrapped file in a text editor before distributing it.

Limitations of the PL/SQL Wrap Utility

- Although wrapping a compilation unit helps to hide the algorithm and makes reverse-engineering hard, Oracle Corporation does not recommend it as a secure method for hiding passwords or table names.
- Because the source code is parsed by the PL/SQL compiler, not by SQL*Plus, you cannot include substitution variables using the SQL*Plus `DEFINE` notation inside the PL/SQL code. You can use substitution variables in other SQL statements that are not obfuscated.
- The wrap utility does not obfuscate the source code for triggers. To hide the workings of a trigger, you can write a one-line trigger that calls a wrapped procedure.
- Some, but not all, comments are removed in wrapped files.
- If your PL/SQL compilation units contain syntax errors, the wrap utility detects and reports them. The wrap utility does not detect semantic errors, such as tables or views that do not exist. Those errors are detected when you run the output file in SQL*Plus.
- The Wrap Utility is upward-compatible between Oracle releases, but is not downward-compatible. For example, you can load files processed by the V8.1.5 wrap utility into a V8.1.6 Oracle database, but you cannot load files processed by the V8.1.6 wrap utility into a V8.1.5 Oracle database.

D

How PL/SQL Resolves Identifier Names

This appendix explains how PL/SQL resolves references to names in potentially ambiguous SQL and procedural statements.

This appendix contains these topics:

What Is Name Resolution?

During compilation, the PL/SQL compiler determines which objects are associated with each name in a PL/SQL subprogram. A name might refer to a local variable, a table, a package, a procedure, a schema, and so on. When a subprogram is recompiled, that association might change if objects have been created or deleted.

A 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 names are not case sensitive:

```
BEGIN
   <<block1>>
   DECLARE
      TYPE Client IS RECORD (...);
      TYPE Customer IS RECORD (...);
   BEGIN
      DECLARE
         client Customer;      -- hides definition of type Client
                               -- in outer scope
         lead1  Client;        -- not allowed; Client resolves to the
                               -- variable client
         lead2  block1.Client; -- OK; refers to type Client
      BEGIN
         NULL;
      END;
```

```
   END;
END;
```

You can still refer to datatype `Client` by qualifying the reference with block label `block1`.

In the following set of `CREATE TYPE` statements, the second statement generates an error. Creating an attribute named `MANAGER` hides the type named `MANAGER`, so the declaration of the second attribute does not work.

```
CREATE TYPE manager AS OBJECT (dept NUMBER);
/
CREATE TYPE person AS OBJECT (manager NUMBER, mgr manager);
/
```

Examples of Qualified Names and Dot Notation

During name resolution, the compiler can encounter various forms of references including simple unqualified names, dot-separated chains of identifiers, indexed components of a collection, and so on. For example:

```
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;
                         -- (9) function call (which is a dot-
```

```
                          --     separated chain of identifiers,
                          --     including schema name used as
                          --     scope qualifier followed by package
                          --     name used as scope qualifier
                          --     followed by function name)
                          --     followed by component selector
                          --     of the returned result followed
                          --     by indexing followed by component
                          --     selector
      n := scott.pkg1.f1.n;
                         -- (10) dot-separated chain of identifiers
                         --      (schema name used as scope qualifier
                         --      followed by package name also used
                         --      as scope qualifier followed by
                         --      function name also used as scope
                         --      qualifier followed by local
                         --      variable name)
      ...
   END f1;

   FUNCTION f2 (q1 NUMBER) RETURN t2 IS
   BEGIN
      ...
   END f2;
END pkg1;
```

Differences in Name Resolution Between SQL and PL/SQL

When the PL/SQL compiler processes a SQL statement, such as a DML statement, it uses the same name-resolution rules as SQL. For example, for a name such as `SCOTT.FOO`, SQL matches objects in the `SCOTT` schema first, then packages, types, tables, and views in the current schema.

PL/SQL uses a different order to resolve names in PL/SQL statements such as assignments and procedure calls. In the case of a name `SCOTT.FOO`, PL/SQL searches first for packages, types, tables, and views named `SCOTT` in the current schema, then for objects in the `SCOTT` schema.

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 no longer refers to an entity in an outer scope:

- The name might now resolve to an entity in an inner scope.
- The program might cause an error, if some part of the identifier is captured in an inner scope and the complete reference cannot be resolved.

If the reference points to a different but valid name, you might not know why the program is acting differently.

In the following example, the reference to `col2` in the inner `SELECT` statement binds to column `col2` in table `tab1` because table `tab2` has no column named `col2`:

```
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER);
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 preceding example, if you add a column named `col2` to table `tab2`:

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

then procedure `proc` is invalidated and recompiled automatically upon next use. However, upon recompilation, the `col2` in the inner `SELECT` statement binds to column `col2` in table `tab2` because `tab2` is in the inner scope. Thus, the reference to `col2` is captured by the addition of column `col2` to table `tab2`.

Using collections and object types can cause more inner capture situations. In the following example, the reference to `s.tab2.a` resolves to attribute `a` of column `tab2` in table `tab1` through table alias `s`, which is visible in the outer scope of the query:

```
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
```

In the preceding example, you might add a column named `a` to table `s.tab2`, which appears in the inner subquery. When the query is processed, an inner capture occurs because the reference to `s.tab2.a` resolves to column `a` of table `tab2` in schema `s`. You can avoid inner captures by following the rules given in "Avoiding Inner Capture in DML Statements" on page D-4. According to those rules, you should revise the query as follows:

```
SELECT * FROM s.tab1 p1
   WHERE EXISTS (SELECT * FROM s.tab2 p2 WHERE p2.x = p1.tab2.a);
```

Same-Scope Capture

In SQL scope, a same-scope capture occurs when a column is added to one of two tables used in a join, so that the same column name exists in both tables. Previously, you could refer to that column name in a join query. To avoid an error, now you must qualify the column name with the table name.

Outer Capture

An outer capture occurs when a name in an inner scope, which once resolved to an entity in an inner scope, is resolved to an entity in an outer scope. SQL and PL/SQL are designed to prevent outer captures. You do not need to take any action to avoid this condition.

Avoiding Inner Capture in DML Statements

You can avoid inner capture in DML statements by following these rules:

- Specify an alias for each table in the DML statement.
- Keep table aliases unique throughout the DML statement.
- Avoid table aliases that match schema names used in the query.
- Qualify each column reference with the table alias.

Qualifying a reference with `schema_name.table_name` does not prevent inner capture if the statement refers to tables with columns of a user-defined object type.

Qualifying References to Object 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. When referencing a table, if you reference the attributes or methods of an object stored in that table, the table name must be accompanied by an alias. As the following examples show, column-qualified references to an attribute or method are not allowed if they are prefixed with a table name:

```
CREATE TYPE t1 AS OBJECT (x NUMBER);
CREATE TABLE tb1 (col t1);
SELECT col.x FROM tb1;                    -- not allowed
SELECT tb1.col.x FROM tb1;                -- not allowed
SELECT scott.tb1.col.x FROM scott.tb1;    -- not allowed
SELECT t.col.x FROM tb1 t;
UPDATE tb1 SET col.x = 10;                -- not allowed
UPDATE scott.tb1 SET scott.tb1.col.x=10;  -- not allowed
UPDATE tb1 t set t.col.x = 1;
DELETE FROM tb1 WHERE tb1.col.x = 10;     -- not allowed
DELETE FROM tb1 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;                     -- not allowed
SELECT REF(o) FROM ot1 o;
SELECT VALUE(ot1) FROM ot1;                   -- not allowed
SELECT VALUE(o) FROM ot1 o;
DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10));  -- not allowed
DELETE FROM ot1 o WHERE VALUE(o) = (t1(10));
UPDATE ot1 SET ot1 = ...                      -- not allowed
UPDATE ot1 o SET o = ....
```

The following ways to insert into an object table are allowed 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 Parameterless Subprograms and Methods

If a subprogram does not take any parameters, you can include an empty set of parentheses or omit the parentheses, both in PL/SQL and in functions called from SQL queries.

For calls to a method that takes no parameters, an empty set of parentheses is optional within PL/SQL scopes but required within SQL scopes.

Name Resolution for SQL Versus PL/SQL

The name-resolution rules for SQL and PL/SQL are similar. You can avoid the few minor differences 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.

E

PL/SQL Program Limits

This appendix discusses the program limits that are imposed by the PL/SQL language.

PL/SQL is based on the programming language Ada. As a result, PL/SQL uses a variant of Descriptive Intermediate Attributed Notation for Ada (DIANA), a tree-structured intermediate language. It is defined using a meta-notation called Interface Definition Language (IDL). DIANA is used internally by compilers and other tools.

At compile time, PL/SQL source code is translated into machine-readable 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

Item	Limit
bind variables passed to a program unit	32K
exception handlers in a program unit	64K
fields in a record	64K
levels of block nesting	255
levels of record nesting	32
levels of subquery nesting	254
levels of label nesting	98
magnitude of a `BINARY_INTEGER` value	2G
magnitude of a `PLS_INTEGER` value	2G
objects referenced by a program unit	64K
parameters passed to an explicit cursor	64K
parameters passed to a function or procedure	64K
precision of a `FLOAT` value (binary digits)	126
precision of a `NUMBER` value (decimal digits)	38

Table E–1 (Cont.) PL/SQL Compiler Limits

Item	Limit
precision of a `REAL` value (binary digits)	63
size of an identifier (characters)	30
size of a string literal (bytes)	32K
size of a `CHAR` value (bytes)	32K
size of a `LONG` value (bytes)	32K-7
size of a `LONG RAW` value (bytes)	32K-7
size of a `RAW` value (bytes)	32K
size of a `VARCHAR2` value (bytes)	32K
size of an `NCHAR` value (bytes)	32K
size of an `NVARCHAR2` value (bytes)	32K
size of a `BFILE` value (bytes)	4G * value of DB_BLOCK_SIZE parameter
size of a `BLOB` value (bytes)	4G * value of DB_BLOCK_SIZE parameter
size of a `CLOB` value (bytes)	4G * value of DB_BLOCK_SIZE parameter
size of an `NCLOB` value (bytes)	4G * value of DB_BLOCK_SIZE parameter

To estimate how much memory a program unit requires, you can query the data dictionary view `user_object_size`. The column `parsed_size` returns the size (in bytes) of the "flattened" DIANA. For example:

```
SQL> SELECT * FROM user_object_size WHERE name = 'PKG1';

NAME    TYPE         SOURCE_SIZE  PARSED_SIZE  CODE_SIZE  ERROR_SIZE
------------------------------------------------------------------
PKG1    PACKAGE               46          165        119           0
PKG1    PACKAGE BODY          82            0        139           0
```

Unfortunately, you cannot estimate the number of DIANA nodes from the parsed size. Two program units with the same parsed size might require 1500 and 2000 DIANA nodes, respectively (because, for example, the second unit contains more complex SQL statements).

When a PL/SQL block, subprogram, package, or object type exceeds a size limit, you get an error such as *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.

F

List of PL/SQL Reserved Words

The words listed in this appendix are reserved by PL/SQL. 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. You should not use them to name schema objects such as columns, tables, or indexes.

ALL*
ALTER*
AND*
ANY*
ARRAY
AS*
ASC*
AT
AUTHID
AVG
BEGIN
BETWEEN*
BINARY_INTEGER
BODY
BOOLEAN
BULK
BY*
CASE
CHAR*
CHAR_BASE
CHECK*
CLOSE
CLUSTER*
COALESCE
COLLECT
COMMENT*
COMMIT
COMPRESS*
CONNECT*
CONSTANT
CREATE*
CURRENT*
CURRVAL
CURSOR
DATE*
DAY
DECLARE
DECIMAL*
DEFAULT*
DELETE*
DESC*
DISTINCT*
DO
DROP*
ELSE*
ELSIF
END
EXCEPTION
EXCLUSIVE*
EXECUTE
EXISTS*
EXIT
EXTENDS
EXTRACT
FALSE
FETCH
FLOAT*
FOR*
FORALL
FROM*
FUNCTION
GOTO
GROUP*
HAVING*
HEAP
HOUR
IF
IMMEDIATE*
IN*
INDEX*
INDICATOR
INSERT*
INTEGER*
INTERFACE
INTERSECT*
INTERVAL
INTO*
IS*
ISOLATION
JAVA
LEVEL*
LIKE*
LIMITED
LOCK*
LONG*
LOOP
MAX
MIN
MINUS*
MINUTE
MLSLABEL*
MOD
MODE*
MONTH
NATURAL
NATURALN
NEW
NEXTVAL
NOCOPY
NOT*
NOWAIT*
NULL*
NULLIF
NUMBER*
NUMBER_BASE
OCIROWID
OF*
ON*
OPAQUE
OPEN
OPERATOR
OPTION*
OR*
ORDER*
ORGANIZATION
OTHERS
OUT
PACKAGE
PARTITION
PCTFREE*
PLS_INTEGER
POSITIVE
POSITIVEN
PRAGMA
PRIOR*
PRIVATE
PROCEDURE
PUBLIC*
RAISE
RANGE
RAW*
REAL
RECORD
REF
RELEASE
RETURN
REVERSE
ROLLBACK
ROW*
ROWID*
ROWNUM*
ROWTYPE
SAVEPOINT
SECOND
SELECT*
SEPARATE
SET*
SHARE*
SMALLINT*
SPACE
SQL
SQLCODE
SQLERRM
START*
STDDEV
SUBTYPE
SUCCESSFUL*
SUM
SYNONYM*
SYSDATE*
TABLE*
THEN*
TIME
TIMESTAMP
TIMEZONE_REGION
TIMEZONE_ABBR
TIMEZONE_MINUTE
TIMEZONE_HOUR
TO*
TRIGGER*
TRUE
TYPE
UID*
UNION*
UNIQUE*
UPDATE*
USE
USER*
VALIDATE*
VALUES*
VARCHAR*
VARCHAR2*
VARIANCE
VIEW*
WHEN
WHENEVER*
WHERE*
WHILE
WITH*
WORK
WRITE
YEAR
ZONE

G

Frequently Asked Questions About PL/SQL

Questions are a burden to others, answers a prison to oneself. —The Prisoner

This appendix contains some frequently asked questions about PL/SQL, and their answers. Where appropriate, the answers link to detailed explanations elsewhere in this book.

Another good source of questions and answers about PL/SQL is the Web site `http://asktom.oracle.com/`. A good source of examples that illustrate the detailed workings of PL/SQL is the PL/SQL area of the Oracle Technology Network site at `http://otn.oracle.com/tech/pl_sql/`.

When Should I Use Bind Variables with PL/SQL?

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

You do need to specify bind variables with dynamic SQL, in clauses like `WHERE` and `VALUES` where you normally use variables. Instead of concatenating literals and variable values into a single string, replace the variables with the names of bind variables (prefixed by a colon) and specify the corresponding PL/SQL variables with the `USING` clause. Using the `USING` clause, instead of concatenating the variables into the string, reduces parsing overhead and lets Oracle reuse the SQL statements.

When Do I Use or Omit the Semicolon with Dynamic SQL?

When building up a single SQL statement in a string, do not include any semicolon at the end (inside the quotation marks).

When building up a PL/SQL anonymous block, include the semicolon at the end of each PL/SQL statement and at the end of the anonymous block. You will have a semicolon right before the end of the string literal, and another right after the closing single quotation mark.

How Can I Use Regular Expressions with PL/SQL?

You can search for regular expressions using the SQL operator `REGEXP_LIKE`.

You can test or manipulate strings using the built-in functions REGEXP_INSTR, REGEXP_REPLACE, and REGEXP_SUBSTR.

Oracle's regular expression features use characters like '.', '*', '^', and '$' that you might be familiar with from UNIX or Perl programming. For multi-language programming, there are also extensions such as '[:lower:]' to match a lowercase letter, instead of '[a-z]' which does not match lowercase accented letters.

How Do I Continue After a PL/SQL Exception?

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

Does PL/SQL Have User-Defined Types or Abstract Data Types?

The PL/SQL term for these things is "object types". To do object-oriented programming, you use a mix of SQL and PL/SQL. You create the types themselves and the tables to hold them using SQL. You write the bodies of the methods in PL/SQL, and you can also manipulate tables of objects and call object methods through PL/SQL.

The best source of information for object-oriented programming with PL/SQL is the book *Oracle Database Application Developer's Guide - Object-Relational Features*.

How Do I Pass a Result Set from PL/SQL to Java or Visual Basic (VB)?

PL/SQL lets you issue a query and return a result set using a cursor variable, also known as a REF CURSOR. See "Using Cursor Variables (REF CURSORs)" on page 6-19.

How Do I Specify Different Kinds of Names with PL/SQL's Dot Notation?

Dot notation is used for identifying record fields, object attributes, and things inside packages or other schemas. When you combine these things, you might need to use expressions with multiple levels of dots, where it isn't always clear what each dot refers to. Here are some of the combinations.

Field or Attribute of a Function Return Value

```
func_name().field_name
func_name().attribute_name
```

Schema Object Owned by Another Schema

```
schema_name.table_name
schema_name.procedure_name()
schema_name.type_name.member_name()
```

Packaged Object Owned by Another User

```
schema_name.package_name.procedure_name()
```

```
schema_name.package_name.record_name.field_name
```

Record Containing an Object Type

```
record_name.field_name.attribute_name
record_name.field_name.member_name()
```

What Can I Do with Objects and Object Types in PL/SQL?

You can create object types by issuing the SQL statement `CREATE TYPE` using dynamic SQL (the `EXECUTE IMMEDIATE` statement).

You can use objects to group related values, as you would do with records, with the added advantage that you can store objects directly in the database. You can also create varrays, nested tables, and associative arrays of objects.

You can write PL/SQL procedures and functions that accept objects as parameters. You can pass objects as parameters to these procedures and functions. You can also return objects from functions.

You can write member procedures and functions for an object type in PL/SQL.

You can create overloaded PL/SQL procedures and functions by defining different versions that accept different subtypes of the same supertypes as parameters.

You can write a PL/SQL application in a completely object-oriented way.

You can write a PL/SQL application in a mostly procedural way, using objects and object types to work around limitations on PL/SQL variables and parameters.

How Do I Create a PL/SQL Procedure?

There are several ways to create different kinds of PL/SQL procedures and functions:

`CREATE PROCEDURE` and `CREATE FUNCTION` statements in SQL.

`CREATE TRIGGER` statement in SQL.

`CREATE PACKAGE` and `CREATE PACKAGE BODY` statements in SQL. The package body can contain both procedures and functions.

`CREATE TYPE` and `CREATE TYPE BODY` statements in SQL. The type body can contain both procedures and functions.

Tip: When using these SQL statements, you might find the `CREATE OR REPLACE` forms more convenient to allow frequent updates without having to drop the original versions.

`PROCEDURE procedure_name IS...` or `FUNCTION function_name RETURN return_type IS...` within the `DECLARE` section of an anonymous block. These procedures and functions only exist for the life of the anonymous block, and are only visible within the block. This technique makes sense if you have a long-running or extensive anonymous block.

`PROCEDURE procedure_name IS...` or `FUNCTION function_name RETURN return_type IS...` within the declarative section of another procedure or function. These procedures and functions are only visible within the outer procedure, so you can use this technique to avoid cluttering up the namespace and prevent unwanted calls from other subprograms.

How Do I Input or Output Data with PL/SQL?

Most PL/SQL I/O is through SQL statements, to store data in database tables or query those tables.

All other PL/SQL I/O is done through APIs that interact with other programs. For example, the `DBMS_OUTPUT` package has procedures such as `PUT_LINE`. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to `DBMS_OUTPUT`. (SQL*Plus does not display `DBMS_OUTPUT` data unless you issue the command `SET SERVEROUTPUT ON` first.)

Other PL/SQL APIs for doing I/O are `HTP` (for displaying output on a web page), `DBMS_PIPE` (for passing information back and forth between PL/SQL and operating-system commands), `UTL_FILE` (for reading and writing operating-system files), `UTL_HTTP` (for communicating with web servers), `UTL_SMTP` (for communicating with mail servers), and `TEXT_IO` (for displaying text from Oracle Forms).

Although some of these APIs can accept input as well as output, there is no built-in language facility for accepting data directly from the keyboard. For that, you can use the `PROMPT` and `ACCEPT` commands in SQL*Plus:

How Do I Perform a Case-Insensitive Query?

You may be familiar with workarounds that allow a query to match `VARCHAR2` values regardless of uppercase and lowercase. For example, you can use the `UPPER()` or `LOWER()` function on the column value, and build a function-based index on the column. Or you can add another column to the table, with a trigger to store an all-uppercase or all-lowercase copy of another column value, and then query this new column.

Now you can make queries case-insensitive (or even accent-insensitive) without any code changes at all. Add `_CI` to the usual value of the `NLS_SORT` initialization parameter, and queries are case-insensitive. Add `_CI` to the usual value of the `NLS_SORT` parameter, and queries are accent-insensitive.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z