PL/SQL

User's Guide and Reference

Release 8.1.5

February 1999

Part No. A67842-01

ORACLE

PL/SQL User's Guide and Reference, Release 8.1.5

Part No. A67842-01

Author: Tom Portfolio

Graphics Artist: Valarie Moore

Contributors: Dave Alpern, Chandrasekharan Iyer, Ervan Darnell, Ken Jacobs, Sanjay Kaluskar, Sanjay Krishnamurthy, Janaki Krishnaswamy, Neil Le, Kannan Muthukkaruppan, Shirish Puranik, Chris Racicot, Ken Rudin, Usha Sangam, Ajay Sethi, Guhan Viswanathan

Contents

2 Fundamentals

3 Control Structures

4 Collections and Records

5 Interaction with Oracle

6 Error Handling

7 Subprograms

8 Packages

9 Object Types

10 Native Dynamic SQL

11 Language Elements

D Name Resolution

E Reserved Words

Index

Send Us Your Comments

PL/SQL User's Guide and Reference, Release 8.1.5

Part No. A67842-01

Oracle Corporation 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 chapter, section, and page number (if available). You can send comments to the Information Development department in the following ways:

- Email: infodev@us.oracle.com
- Fax: (650) 506-7228 Attn: Oracle Server Documentation
- Letter:
  Oracle Corporation
  Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065 USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

Preface

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

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

Major Topics

Audience

Anyone developing PL/SQL-based applications for Oracle8*i* will benefit from reading this guide. Written especially for programmers, this comprehensive treatment of PL/SQL will also be of value to systems analysts, project managers, and others interested in database applications. To use this guide effectively, you need a working knowledge of Oracle8*i*, SQL, and a 3GL such as Ada, C, or COBOL.

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

How This Guide Is Organized

The *PL/SQL User's Guide and Reference* has 11 chapters and 5 appendices. Chapters 1 through 10 introduce you to PL/SQL and show you how to use its many features. Chapter 11 serves as a reference to PL/SQL commands, syntax, and semantics. Appendices A through E provide sample programs, supplementary technical information, and a list of reserved words.

Chapter 1, "Overview" This chapter surveys the main features of PL/SQL and points out the advantages they offer. It also acquaints you with the basic concepts behind PL/SQL and the general appearance of PL/SQL programs.

Chapter 2, "Fundamentals" This chapter focuses on the small-scale aspects of PL/SQL. It discusses lexical units, scalar datatypes, user-defined subtypes, data conversion, expressions, assignments, block structure, declarations, and scope.

Chapter 3, "Control Structures" This chapter shows you how to structure the flow of control through a PL/SQL program. It describes conditional, iterative, and sequential control. You learn how to apply simple but powerful control structures such as `IF-THEN-ELSE` and `WHILE-LOOP`.

Chapter 4, "Collections and Records" This chapter focuses on the composite datatypes `TABLE`, `VARRAY`, and `RECORD`. You learn how to reference and manipulate whole collections of data, and how to treat related but dissimilar data as a logical unit. You also learn how to improve performance by bulk-binding collections.

Chapter 5, "Interaction with Oracle" This chapter shows you how PL/SQL supports the SQL commands, functions, and operators that let you manipulate Oracle data. You also learn how to manage cursors, process transactions, and safeguard your database.

Chapter 6, "Error Handling" This chapter provides an in-depth discussion of error reporting and recovery. You learn how to detect and handle errors using PL/SQL exceptions.

Chapter 7, "Subprograms" This chapter shows you how to write and use subprograms. It discusses procedures, functions, forward declarations, actual and formal parameters, positional and named notation, parameter modes, the NOCOPY compiler hint, parameter default values, aliasing, overloading, invoker rights, and recursion.

Chapter 8, "Packages" This chapter shows you how to bundle related PL/SQL types, items, and subprograms into a package. Once written, your general-purpose package is compiled, then stored in an Oracle database, where its contents can be shared by many applications.

Chapter 9, "Object Types" This chapter introduces you to object-oriented programming based on object types, which provide abstract templates for real-world objects. You learn how to define object types and manipulate objects.

Chapter 10, "Native Dynamic SQL" This chapter shows you how to use dynamic SQL, an advanced programming technique that makes your applications more flexible and versatile. You learn two simple ways to write programs that can build and process SQL statements "on the fly" at run time.

Chapter 11, "Language Elements" This chapter uses syntax diagrams to show how commands, parameters, and other language elements are sequenced to form PL/SQL statements. Also, it provides usage notes and short examples to help you become fluent in PL/SQL quickly.

Appendix A, "Sample Programs" This appendix provides several PL/SQL programs to guide you in writing your own. The sample programs illustrate important concepts and features.

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

Appendix C, "PL/SQL Wrapper" This appendix shows you how to run the PL/SQL Wrapper, a stand-alone utility that enables you to deliver PL/SQL applications without exposing your source code.

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

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

Notational Conventions

This guide follows these conventions:

Convention	Meaning
Italic	Italic font denotes terms being defined for the first time, words being emphasized, error messages, and book titles.
`Courier`	Courier font denotes PL/SQL code, keywords, program names, file names, and path names.

PL/SQL code examples follow these conventions:

Convention	Meaning
`--`	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.
`...`	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.
`lower case`	Lower case is used for names of constants, variables, cursors, exceptions, subprograms, and packages.
`UPPER CASE`	Upper case is used for keywords, names of predefined exceptions, and names of supplied PL/SQL packages.
`Mixed Case`	Mixed case is used for names of user-defined datatypes and subtypes. The names of user-defined types begin with an upper-case letter.

Syntax definitions use a simple variant of Backus-Naur Form (BNF) that includes the following symbols:

Symbol	Meaning
`[ ]`	Brackets enclose optional items.
`{ }`	Braces enclose items only one of which is required.
`|`	A vertical bar separates alternatives within brackets or braces.
`...`	An ellipsis shows that the preceding syntactic element can be repeated.
`delimiters`	Delimiters other than brackets, braces, vertical bars, and ellipses must be entered as shown.

Sample Database Tables

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

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

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

Sample Data

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

```
DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON
```

```
EMPNO ENAME   JOB          MGR  HIREDATE    SAL   COMM  DEPTNO
----- ------- --------- ------ --------- ------ ------ -------
 7369 SMITH   CLERK       7902 17-DEC-80    800             20
 7499 ALLEN   SALESMAN    7698 20-FEB-81   1600    300      30
 7521 WARD    SALESMAN    7698 22-FEB-81   1250    500      30
 7566 JONES   MANAGER     7839 02-APR-81   2975             20
 7654 MARTIN  SALESMAN    7698 28-SEP-81   1250   1400      30
 7698 BLAKE   MANAGER     7839 01-MAY-81   2850             30
 7782 CLARK   MANAGER     7839 09-JUN-81   2450             10
 7788 SCOTT   ANALYST     7566 19-APR-87   3000             20
 7839 KING    PRESIDENT        17-NOV-81   5000             10
 7844 TURNER  SALESMAN    7698 08-SEP-81   1500             30
 7876 ADAMS   CLERK       7788 23-MAY-87   1100             20
 7900 JAMES   CLERK       7698 03-DEC-81    950             30
 7902 FORD    ANALYST     7566 03-DEC-81   3000             20
 7934 MILLER  CLERK       7782 23-JAN-82   1300             10
```

Your Comments Are Welcome

We in Information Development appreciate your comments and suggestions. In fact, your opinions are the most important feedback we receive. We encourage you to use the Reader's Comment Form at the front of this guide. You can also send comments to us by

- Email: infodev@us.oracle.com
- Fax: (650) 506-7228 Attn: Oracle Server Documentation
- Letter:

  Oracle Corporation
  Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065 USA

1

Overview

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

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

Major Topics

Main Features

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

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

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

Block Structure

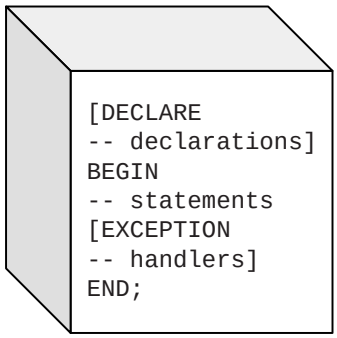
PL/SQL is a *block-structured* language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or subproblem to be solved. Thus, PL/SQL supports the divide-and-conquer approach to problem solving called *stepwise refinement*.

A block (or sub-block) lets you group logically related declarations and statements. That way, you can place declarations close to where they are used. The declarations are local to the block and cease to exist when the block completes.

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

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

Figure 1–1 Block Structure



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

Variables and Constants

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

Declaring Variables

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

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

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

Assigning Values to a Variable

You can assign values to a variable in two 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 to the right. Some examples follow:

```
tax := price * tax_rate;
bonus := current_salary * 0.10;
amount := TO_NUMBER(SUBSTR('750 dollars', 1, 3));
valid := FALSE;
```

The second way to assign values to a variable is to select or fetch database values into it. In the following example, you have Oracle compute a 10% bonus when you select the salary of an employee:

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

Then, you can use the variable bonus in another computation or insert its value into a database table.

Declaring Constants

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

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

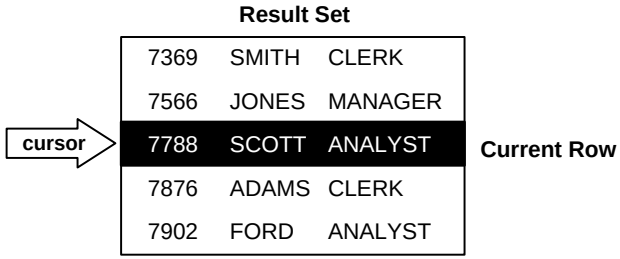
Cursors

Oracle uses work areas to execute SQL statements and store processing information. A PL/SQL construct called a *cursor* lets you name a work area and access its stored information. There are two kinds of cursors: *implicit* and *explicit*. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually. An example follows:

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

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

Figure 1–2 Query Processing



Multi-row query processing is somewhat like file processing. For example, a COBOL program opens a file, processes records, then closes the file. Likewise, a PL/SQL program opens a cursor, processes rows returned by a query, then closes the cursor. Just as a file pointer marks the current position in an open file, a cursor marks the current position in a result set.

You use the `OPEN`, `FETCH`, and `CLOSE` statements to control a cursor. The `OPEN` statement executes the query associated with the cursor, identifies the result set, and positions the cursor before the first row. The `FETCH` statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the `CLOSE` statement disables the cursor.

Cursor FOR Loops

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

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

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

Cursor Variables

Like a cursor, a cursor variable points to the current row in the result set of a multi-row query. But, unlike a cursor, a cursor variable can be opened for any type-compatible query. It is not tied to a specific query. Cursor variables are true PL/SQL variables, to which you can assign new values and which you can pass to subprograms stored in an Oracle database. This gives you more flexibility and a convenient way to centralize data retrieval.

Typically, you open a cursor variable by passing it to a stored procedure that declares a cursor variable as one of its formal parameters. The following procedure opens the cursor variable generic_cv for the chosen query:

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

Attributes

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

%TYPE

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

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

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

%ROWTYPE

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The %ROWTYPE attribute provides a record type that represents a row in a table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable.

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

```
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:

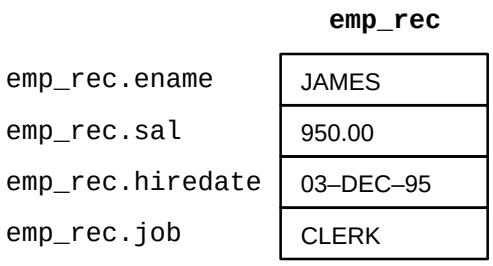
```
DECLARE
   CURSOR c1 IS
      SELECT ename, sal, hiredate, job FROM emp;
   emp_rec c1%ROWTYPE;  -- declare record variable that represents
                        -- a row fetched from the emp table
```

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. Figure 1–3 shows how the result might appear.

Figure 1–3 %ROWTYPE Record



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, FOR-LOOP, WHILE-LOOP, EXIT-WHEN, and GOTO. Collectively, these statements can handle any situation.

Conditional Control

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

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

```
-- 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;
```

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. This can improve performance and simplify the integrity checks built into Oracle Forms applications.

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, suppose that you are a manufacturer of custom-made cars and that each car has a serial number. To keep track of which customer buys each car, you might use the following FOR loop:

```
FOR i IN 1..order_qty LOOP
   UPDATE sales SET custno = customer_id
      WHERE serial_num = serial_num_seq.NEXTVAL;
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 $4000 and is higher in the chain of command than employee 7902:

```
-- available online in file 'examp3'
DECLARE
   salary         emp.sal%TYPE;
   mgr_num        emp.mgr%TYPE;
   last_name      emp.ename%TYPE;
   starting_empno CONSTANT NUMBER(4) := 7902;
BEGIN
   SELECT sal, mgr INTO salary, mgr_num FROM emp
      WHERE empno = starting_empno;
   WHILE salary < 4000 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;
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;
```

Modularity

Modularity lets you break an application down into manageable, well-defined logic modules. Through successive refinement, you can reduce a complex problem to a set of simple problems that have easy-to-implement solutions. PL/SQL meets this need with *program units*. Besides blocks and subprograms, PL/SQL provides the package, which allows you to group related program items into larger units.

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
   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
   WHEN comm_missing THEN
      ...
END award_bonus;
```

When called, this procedure accepts an employee number. It uses the number to select the employee's commission from a database table and, at the same time, compute a 15% bonus. Then, it checks the bonus amount. If the bonus is null, an exception is raised; otherwise, the employee's payroll record is updated.

Packages

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

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

In the following example, you package two employment procedures:

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

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

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

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

Data Abstraction

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

Collections

The collection types TABLE and VARRAY allow you to declare nested tables and variable-size arrays (varrays for short). A collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection.

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

```
DECLARE
   TYPE Staff IS TABLE OF Employee;
   staffer Employee;
   FUNCTION new_hires (hiredate DATE) RETURN Staff IS
   BEGIN ... END;
```

```
BEGIN
   staffer := new_hires('10-NOV-98')(5);
   ...
END;
```

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

Records

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

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

Consider the following example:

```
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));
```

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

Object Types

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

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

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

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

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

Information Hiding

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

Algorithms

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

Data Structures

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

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

Error Handling

PL/SQL makes it easy to detect and process predefined and user-defined error conditions called *exceptions*. When an error occurs, an exception is *raised*. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. To handle raised exceptions, you write separate routines called *exception handlers*.

Predefined exceptions are raised implicitly by the runtime system. For example, if you try to divide a number by zero, PL/SQL raises the predefined exception `ZERO_DIVIDE` automatically. You must raise user-defined exceptions explicitly with the `RAISE` statement.

You can define exceptions of your own in the declarative part of any PL/SQL block or subprogram. In the executable part, you check for the condition that needs special attention. If you find that the condition exists, you execute a `RAISE` statement. In the example below, you compute the bonus earned by a salesperson. The bonus is based on salary and commission. So, if the commission is null, you raise the exception `comm_missing`.

```
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 exception
```

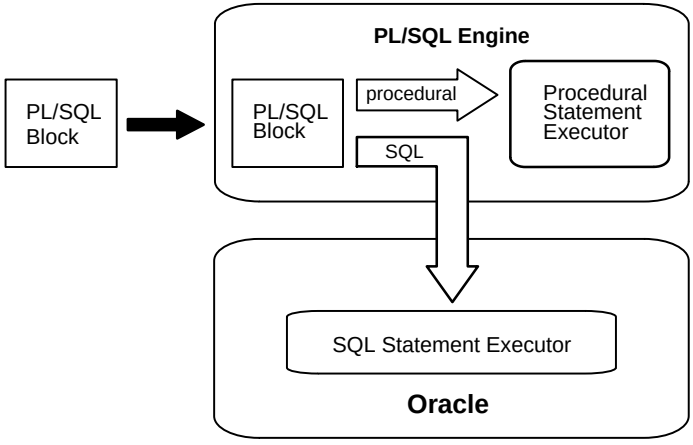
Architecture

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

- the Oracle server
- Oracle tools

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

Figure 1–4 PL/SQL Engine



In the Oracle Server

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

Anonymous Blocks

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

Stored Subprograms

Subprograms can be compiled separately and stored permanently in an Oracle database, ready to be executed. A subprogram explicitly CREATEd using an Oracle tool is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a schema object, which can be referenced by any number of applications connected to that database.

Stored subprograms defined within a package are called *packaged* subprograms. Those defined independently are called *stand-alone* subprograms. Those defined within another subprogram or within a PL/SQL block are called *local* subprograms, which cannot be referenced by other applications and exist only for the convenience of the enclosing block.

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

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

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

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

Database Triggers

A database trigger is a stored subprogram associated with a table. You can have Oracle automatically fire the database trigger before or after an INSERT, UPDATE, or DELETE statement affects the table. One of the many uses for database triggers is to audit data modifications. For example, the following database 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;
```

You can use all the SQL data manipulation statements and any procedural statement in the executable part of a database trigger.

In Oracle Tools

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

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

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

Advantages of PL/SQL

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

- support for SQL
- support for object-oriented programming
- better performance
- higher productivity
- full portability
- tight integration with Oracle
- security

Support for SQL

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

SQL is non-procedural, meaning that you can state what you want done without stating how to do it. Oracle determines the best way to carry out your request. There is no necessary connection between consecutive statements because Oracle executes SQL statements one at a time.

PL/SQL lets you use all the SQL data manipulation, cursor control, and transaction control commands, as well as all the SQL functions, operators, and pseudocolumns. So, you can manipulate Oracle data flexibly and safely. Also, PL/SQL fully supports SQL datatypes. That reduces the need to convert data passed between your applications and the database.

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

Support for Object-Oriented Programming

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

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

In addition, object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. That helps your programs better reflect the world they are trying to simulate.

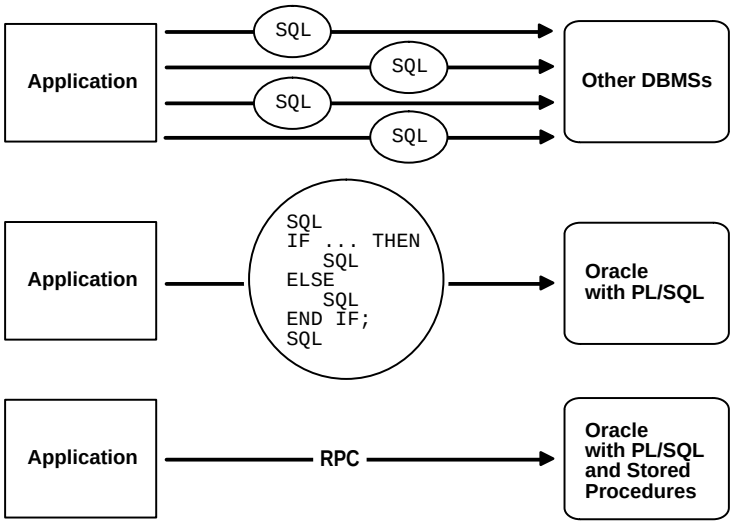
Better Performance

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

However, with PL/SQL, an entire block of statements can be sent to Oracle at one time. This can drastically reduce communication between your application and Oracle. As Figure 1–5 shows, if your application is database intensive, you can use PL/SQL blocks and subprograms to group SQL statements before sending them to Oracle for execution.

PL/SQL stored procedures are compiled once and stored in executable form, so procedure calls are quick and efficient. Also, stored procedures, which execute in the server, can be invoked over slow network connections with a single call. That reduces network traffic and improves round-trip response times. Executable code is automatically cached and shared among users. That lowers memory requirements and invocation overhead.

Figure 1–5 PL/SQL Boosts Performance



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

Higher Productivity

PL/SQL adds functionality to non-procedural tools such as Oracle Forms and Oracle Reports. With PL/SQL in these tools, you can use familiar procedural constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger. You need not use multiple trigger steps, macros, or user exits. Thus, PL/SQL increases productivity by putting better tools in your hands.

Moreover, PL/SQL is the same in all environments. As soon as you master PL/SQL with one Oracle tool, you can transfer your knowledge to other tools, and so multiply the productivity gains. For example, scripts written with one tool can be used by other tools.

Full Portability

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

Tight Integration with SQL

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

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

Security

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

Furthermore, you can restrict access to Oracle data by allowing users to manipulate it only through stored procedures that execute with their definer's privileges. For example, you can grant users access to a procedure that updates a table, but not grant them access to the table itself.

2

Fundamentals

There are six essentials in painting. The first is called spirit; the second, rhythm; the third, thought; the fourth, scenery; the fifth, the brush; and the last is the ink. —Ching Hao

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

Major Topics

Character Set

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

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

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

Lexical Units

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

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

For example, the line

```
bonus := salary * 0.10;  -- compute bonus
```

contains the following lexical units:

- identifiers `bonus` and `salary`
- compound symbol `:=`
- simple symbols `*` and `;`
- numeric literal `0.10`
- comment `-- compute bonus`

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 illegal because the reserved words `END` and `IF` are joined:

```
IF x > y THEN high := x; ENDIF;  -- illegal
```

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

```
count : = count + 1;  -- illegal
```

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

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

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

Delimiters

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

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

Compound symbols consist of two characters. A list follows:

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 illegal, as the following examples show:

```
mine&yours    -- illegal ampersand
debit-amount  -- illegal hyphen
on/off        -- illegal slash
user id       -- illegal space
```

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

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

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

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

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

```
lastname
last_name
```

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

Reserved Words

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

```
DECLARE
   end BOOLEAN;  -- illegal; causes compilation error
```

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

```
DECLARE
   end_of_game BOOLEAN;  -- legal
```

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

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

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

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

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

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

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

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

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

Literals

A *literal* is an explicit numeric, character, string, or Boolean value not represented by an identifier. The numeric literal 147 and the Boolean literal FALSE are examples.

Numeric Literals

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

```
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 must be 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
```

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.

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

```
'Don''t leave without saving your work.'
```

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

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

Boolean Literals

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

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

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

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

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

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

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

Multi-line

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

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

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

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

Restrictions

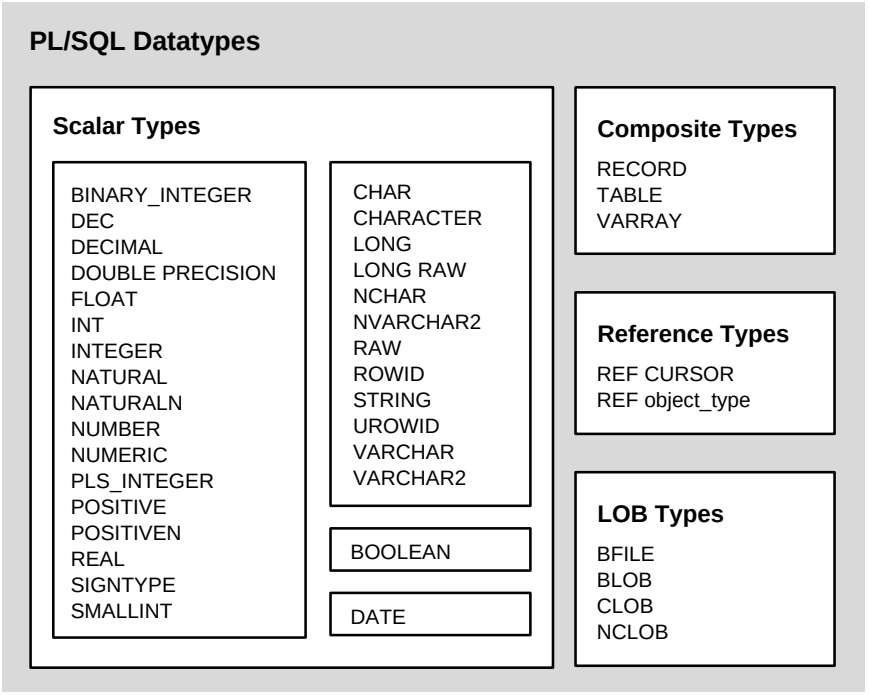
You cannot nest comments. Also, you cannot use single-line comments in a PL/SQL block that will be processed dynamically by an Oracle Precompiler program because end-of-line characters are ignored. As a result, single-line comments extend to the end of the block, not just to the end of a line. So, use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

Datatypes

Every constant and variable has a *datatype*, which specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined datatypes. A *scalar* type has no internal components. A *composite* type has internal components that can be manipulated individually. A *reference* type holds values, called *pointers*, that designate other program items. A LOB type holds values, called lob locators, that specify the location of large objects (graphic images for example) stored out-of-line.

Figure 2–1 shows the predefined datatypes available for your use. The scalar types fall into four families, which store number, character, Boolean, and date/time data, respectively.

Figure 2–1 Built-in Datatypes



This section discusses the scalar types and LOB types. The composite types are discussed in Chapter 4. The reference types are discussed in Chapter 5 and Chapter 9.

Number Types

Number types allow you to 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 -2147483647 .. 2147483647. Like PLS_INTEGER values, BINARY_INTEGER values require less storage than NUMBER values. However, most BINARY_INTEGER operations are slower than PLS_INTEGER operations.

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

```
NATURAL
NATURALN
POSITIVE
POSITIVEN
SIGNTYPE
```

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

NUMBER

You use the NUMBER datatype to store fixed-point or floating-point numbers of virtually any size. 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 -2147483647 .. 2147483647. 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 better performance, use PLS_INTEGER for all calculations that fall within its magnitude range.

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

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

Character Types

Character types allow you to 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 length up to 32767 bytes. The syntax follows:

```
CHAR[(maximum_length)]
```

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

If you do not specify a maximum length, it defaults to 1. Remember, you specify the maximum length in bytes, not characters. So, if a CHAR(n) variable stores multi-byte characters, its maximum length is less than n characters. The maximum width of a CHAR database column is 2000 bytes. So, you cannot insert CHAR values longer than 2000 bytes into a CHAR column.

You can insert any CHAR(n) value into a LONG database column because the maximum width of a LONG column is 2147483647 bytes or 2 gigabytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG column into a CHAR(n) variable.

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 length 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 length of a LONG RAW value is 32760 bytes.

You can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2147483647 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 2147483647 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 *Oracle8i SQL Reference*.

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, Net8 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 length up to 32767 bytes. The syntax follows:

```
RAW(maximum_length)
```

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

The maximum width of a RAW database column is 2000 bytes. So, you cannot insert RAW values longer than 2000 bytes into a RAW column. You can insert any RAW value into a LONG RAW database column because the maximum width of a LONG RAW column is 2147483647 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 UROWID 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 UROWID 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.

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 5-49.

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

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

A logical rowid can include a *guess*, which identifies the block location of a row at the time the guess is made. Bypassing 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. However, when fetching from highly volatile tables, it's a good idea to use rowids without guesses.

Note: To manipulate rowids, you can use the supplied package DBMS_ROWID. For more information, see *Oracle8i Supplied Packages 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 length up to 32767 bytes. The syntax follows:

```
VARCHAR2(maximum_length)
```

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

The VARCHAR2 datatype involves a trade-off between memory use and efficiency. For a VARCHAR2(>= 2000) variable, PL/SQL dynamically allocates only enough memory to hold the actual value. However, for a VARCHAR2(< 2000) variable, PL/SQL preallocates enough memory to hold a maximum-size value. So, for example, if you assign the same 500-byte value to a VARCHAR2(2000) variable and to a VARCHAR2(1999) variable, the latter uses 1499 bytes more memory.

Remember, you specify the maximum length of a VARCHAR2(n) variable in bytes, not characters. So, if a VARCHAR2(n) variable stores multi-byte characters, its maximum length is less than n characters. The maximum width of a VARCHAR2 database column is 4000 bytes. Therefore, you cannot insert VARCHAR2 values longer than 4000 bytes into a VARCHAR2 column.

You can insert any VARCHAR2(n) value into a LONG database column because the maximum width of a LONG column is 2147483647 bytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG column into a VARCHAR2(n) variable.

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

```
STRING
VARCHAR
```

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

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

NLS Character Types

Although the widely used 7- or 8-bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain thousands of characters. These languages require 16 bits (two bytes) to represent each character. How does Oracle deal with such dissimilar languages?

Oracle provides National Language Support (NLS), which lets you process single-byte and multi-byte character data and convert between character sets. It also lets your applications run in different language environments.

With NLS, number and date formats adapt automatically to the language conventions specified for a user session. Thus, NLS allows users around the world to interact with Oracle in their native languages. For more information about NLS, see *Oracle8i National Language Support Guide*.

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 NLS data. The datatypes NCHAR and NVARCHAR2 store character strings formed from the national character set.

NCHAR

You use the NCHAR datatype to store fixed-length (blank-padded if necessary) NLS character data. How the data is represented internally depends on the national character set, which might use a fixed-width encoding such as US7ASCII or a variable-width encoding such as JA16SJIS.

The NCHAR datatype takes an optional parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows:

```
NCHAR[(maximum_length)]
```

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

If you do not specify a maximum length, it defaults to 1. How you specify the maximum length depends on the national character set. For fixed-width character sets, you specify the maximum length in characters. For variable-width character sets, you specify it in bytes. In the following example, the character set is JA16EUCFIXED, which is fixed-width, so you specify the maximum length in characters:

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

The maximum width of an NCHAR database column is 2000 bytes. So, you cannot insert NCHAR values longer than 2000 bytes into an NCHAR column. Remember, for fixed-width, multi-byte character sets, you cannot insert NCHAR values longer than the number of characters that fit in 2000 bytes.

If the NCHAR value is shorter than the defined width of the NCHAR column, Oracle blank-pads the value to the defined width. You cannot insert CHAR values into an NCHAR column. Likewise, you cannot insert NCHAR values into a CHAR column.

NVARCHAR2

You use the NVARCHAR2 datatype to store variable-length NLS character data. How the data is represented internally depends on the national character set, which might use a fixed-width encoding such as WE8EBCDIC37C or a variable-width encoding such as JA16DBCS.

The NVARCHAR2 datatype takes a required parameter that specifies a maximum length up to 32767 bytes. The syntax follows:

```
NVARCHAR2(maximum_length)
```

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

How you specify the maximum length depends on the national character set. For fixed-width character sets, you specify the maximum length in characters. For variable-width character sets, you specify it in bytes. In the following example, the character set is JA16SJIS, which is variable-width, so you specify the maximum length in bytes:

```
my_string NVARCHAR2(200);  -- maximum length is 200 bytes
```

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. Remember, for fixed-width, multi-byte character sets, you cannot insert NVARCHAR2 values longer than the number of characters that fit in 4000 bytes.

You cannot `insert`VARCHAR2 values into an NVARCHAR2 column. Likewise, you cannot insert NVARCHAR2 values into a VARCHAR2 column.

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 values, called *lob locators*, that specify the location of 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 retrieve a BLOB column value, only a locator is returned. Locators cannot span transactions or sessions. So, you cannot save a locator in a PL/SQL variable during one transaction or session, then use it in another transaction or session. To manipulate LOBs, use the supplied package DBMS_LOB. For more information about LOBs and package DBMS_LOB, see *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

BFILE

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

BFILEs are read-only. 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 replicated. Changes made by package DBMS_LOB or the OCI can be committed or rolled back. However, BLOB locators cannot span transactions or sessions.

CLOB

You use the CLOB datatype to store large blocks of single-byte 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 replicated. Changes made by package DBMS_LOB or the OCI can be committed or rolled back. However, CLOB locators cannot span transactions or sessions.

NCLOB

You use the NCLOB datatype to store large blocks of multi-byte 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 replicated. Changes made by package DBMS_LOB or the OCI can be committed or rolled back. However, NCLOB locators cannot span transactions or sessions.

Other Types

The following types allow you to store and manipulate logical values and date/time values.

BOOLEAN

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

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

DATE

You use the DATE datatype to store fixed-length date/time values. DATE values 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.

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

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

You can add and subtract dates. For example, the following statement returns the number of days since an employee was hired:

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

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

For more information about date functions and format models, see *Oracle8i SQL Reference*.

User-Defined Subtypes

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

Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables. PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtype CHARACTER, as follows:

```
SUBTYPE CHARACTER IS CHAR;
```

The subtype CHARACTER specifies the same set of values as its base type CHAR. Thus, CHARACTER is an *unconstrained* 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 [NOT NULL];
```

where subtype_name is a type specifier used in subsequent declarations and base_type is any scalar or user-defined PL/SQL type. To specify base_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, cursor variable, or database table. 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
   CURSOR c1 IS SELECT * FROM dept;
   SUBTYPE DeptFile IS c1%ROWTYPE;      -- based on cursor rowtype
```

However, you cannot specify a size constraint on the base type. For example, the following definitions are illegal:

```
DECLARE
   SUBTYPE Accumulator IS NUMBER(7,2); -- illegal; must be NUMBER
   SUBTYPE Delimiter IS CHAR(1);       -- illegal; must be CHAR
```

Although you cannot define size-constrained subtypes directly, you can use a simple workaround to define them indirectly. Just declare a size-constrained variable, then use %TYPE to provide its datatype, as shown in the following example:

```
DECLARE
   temp VARCHAR2(15);
   SUBTYPE Word IS temp%TYPE; -- maximum size of Word is 15
```

Likewise, if you define a subtype using %TYPE to provide the datatype of a database column, the subtype inherits the size constraint (if any) of the column. However, the subtype does *not* inherit other kinds of constraints such as NOT NULL.

Using Subtypes

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

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

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

```
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 Scale to storing integers in the range -9 .. 9. If your program tries to store a number outside that range in a Scale variable, PL/SQL raises an exception.

```
DECLARE
   temp NUMBER(1,0);
   SUBTYPE Scale IS temp%TYPE;
```

```
   x_axis Scale;  -- magnitude range is -9 .. 9
   y_axis Scale;
BEGIN
   x_axis := 10;  -- raises VALUE_ERROR
```

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;
```

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

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

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;
   SUBTYPE Text IS VARCHAR2;
   verb     Word;
   sentence Text;
BEGIN
   ...
   sentence := verb;
```

Datatype Conversion

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

Explicit Conversion

To convert values from one datatype to another, you use built-in functions. For example, to convert a CHAR value to a DATE or NUMBER value, you use the function TO_DATE or TO_NUMBER, respectively. Conversely, to convert a DATE or NUMBER value to a CHAR value, you use the function TO_CHAR. For more information about these functions, see *Oracle8i SQL Reference*.

Implicit Conversion

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

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

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

Likewise, before assigning the value of a variable to a database column, PL/SQL will, if necessary, convert the value from the datatype of the variable to the datatype of the target column. If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. In such cases, you must use a datatype conversion function. Table 2–1 shows which implicit conversions PL/SQL can do.

Table 2–1 Implicit Conversions

	BIN_INT	CHAR	DATE	LONG	NUMBER	PLS_INT	RAW	UROWID	VARCHAR2
BIN_INT		X		X	X	X			X
CHAR	X		X	X	X	X	X	X	X
DATE		X		X					X
LONG		X					X		X
NUMBER	X	X		X		X			X
PLS_INT	X	X		X	X				X
RAW		X		X					X
UROWID		X							X
VARCHAR2	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.

Implicit versus Explicit Conversion

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

DATE Values

When you select a DATE column value into a CHAR or VARCHAR2 variable, PL/SQL must convert the internal binary value to a character value. So, 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, you must call TO_CHAR with a format mask.

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

RAW and LONG RAW Values

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

A conversion is also necessary when you insert a CHAR or VARCHAR2 value into a RAW or LONG RAW column. Each pair of characters in the variable must represent the hexadecimal equivalent of a binary byte. If either character does not represent the hexadecimal equivalent of a nibble, PL/SQL raises an exception.

NLS Values

When passed an uppercase character set name, the built-in function NLS_CHARSET_ID returns the corresponding character set ID number. Conversely, when passed a character set ID number, the function NLS_CHARSET_NAME returns the corresponding character set name.

If you pass the value 'CHAR_CS' or 'NCHAR_CS' to NLS_CHARSET_ID, it returns the database or national character set ID number, respectively. For a list of character set names, see *Oracle8i National Language Support Guide*.

Declarations

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

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

A couple of examples follow:

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

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

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

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

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

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

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

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

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

Using DEFAULT

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

```
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. A couple of examples follow:

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

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

Using NOT NULL

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

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

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

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

Recall that the subtypes NATURALN and POSITIVEN are predefined as NOT NULL. For instance, the following declarations are equivalent:

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

In NATURALN and POSITIVEN declarations, the type specifier must be followed by an initialization clause. Otherwise, you get a compilation error. For example, the following declaration is illegal:

```
line_items POSITIVEN;  -- illegal; not initialized
```

Using %TYPE

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

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

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

```
balance         NUMBER(7,2);
minimum_balance balance%TYPE := 10.00;
```

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

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

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

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

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

Using %ROWTYPE

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

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

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

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

```
BEGIN
   SELECT * INTO emp_rec FROM emp WHERE ...
```

The column values returned by the SELECT statement are stored in fields. To reference a field, you use dot notation. For example, you might reference the deptno field as follows:

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

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

```
emp_rec.ename := 'JOHNSON';
emp_rec.sal := emp_rec.sal * 1.15;
```

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

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

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

Aggregate Assignment

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

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

However, because dept_rec2 is based on a table and dept_rec3 is based on a cursor, the following assignment is illegal:

```
dept_rec2 := dept_rec3;  -- illegal
```

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

```
DECLARE
   dept_rec dept%ROWTYPE;
   ...
BEGIN
   SELECT deptno, dname, loc INTO dept_rec FROM dept
      WHERE deptno = 30;
```

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

```
record_name := (value1, value2, value3, ...);  -- illegal
```

Although you can retrieve entire records, you cannot insert or update them. For example, the following statement is illegal:

```
INSERT INTO dept VALUES (dept_rec);  -- illegal
```

Using Aliases

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

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

Restrictions

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

```
maxi INTEGER := 2 * mini;  -- illegal
mini INTEGER := 15;
```

However, PL/SQL does allow the forward declaration of subprograms. For more information, see "Forward Declarations" on page 7-9.

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

```
i, j, k SMALLINT;  -- illegal
```

The legal version follows:

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

Naming Conventions

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

```
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, stand-alone subprograms, and packages. However, you cannot create synonyms for items declared within subprograms or packages. That includes constants, variables, cursors, cursor variables, exceptions, and packaged subprograms.

Scoping

Within the same scope, all declared identifiers must be unique. So, even if their datatypes differ, variables and parameters cannot share the same name. For example, two of the following declarations are illegal:

```
DECLARE
   valid_id BOOLEAN;
   valid_id VARCHAR2(5);  -- illegal duplicate identifier
   FUNCTION bonus (valid_id IN INTEGER) RETURN REAL IS ...
                          -- illegal triplicate identifier
```

For the scoping rules that apply to identifiers, see "Scope and Visibility" on page 2-37.

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;  -- same as zip_code
```

Name Resolution

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

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

In such cases, to avoid ambiguity, prefix the names of local variables and formal parameters with my_, as follows:

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

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

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

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

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

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

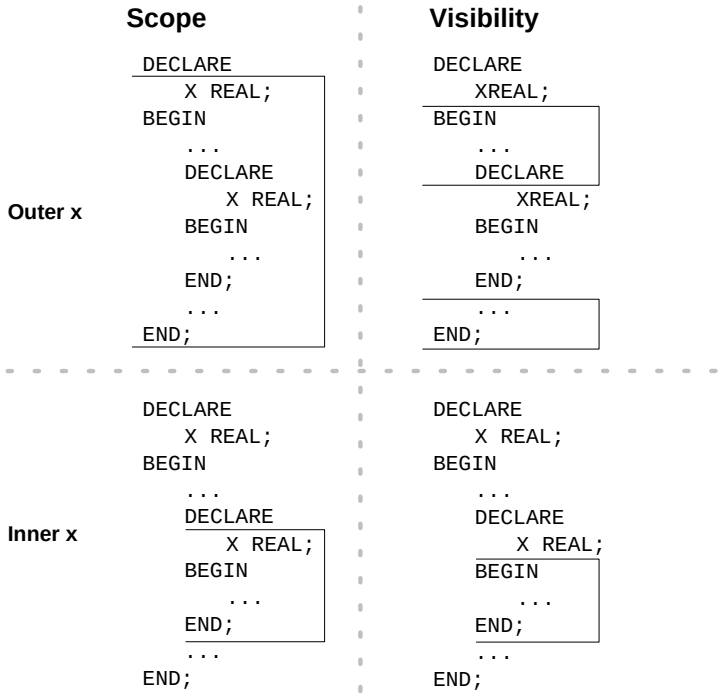
Scope and Visibility

References to an identifier are resolved according to its scope and visibility. The *scope* of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier. An identifier is *visible* only in the regions from which you can reference the identifier using an unqualified name. Figure 2–2 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 2–2 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, as the following example shows:

```
<<outer>>
DECLARE
   birthdate DATE;
BEGIN
   DECLARE
      birthdate DATE;
   BEGIN
      ...
      IF birthdate = outer.birthdate THEN ...
```

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 ...
```

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

Assignments

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

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

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

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

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

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

Boolean Values

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

```
DECLARE
   done BOOLEAN;
```

the following statements are legal:

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

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

```
done := (count > 500);
```

Database Values

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

```
DECLARE
   my_empno emp.empno%TYPE;
   my_ename emp.ename%TYPE;
   wages    NUMBER(7,2);
BEGIN
   ...
   SELECT ename, sal + comm
      INTO last_name, wages FROM emp
      WHERE empno = emp_id;
```

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

Expressions and Comparisons

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

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

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

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

Operator Precedence

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

Table 2–2 Order of Operations

Operator	Operation
**, NOT	exponentiation, logical negation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	comparison
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 2–3. AND and OR are binary operators; NOT is a unary operator.

Table 2–3 Logic Truth Table

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	
TRUE	NULL	NULL	TRUE	
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	
FALSE	NULL	FALSE	NULL	
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	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. It follows that if you apply the NOT operator to a null, the result is also indeterminate. Be careful. Nulls can cause unexpected results; see "Handling Nulls" on page 2-48.

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 allows you to write expressions that might otherwise cause an error. Consider the following OR expression:

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

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

Comparison Operators

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

```
IF quantity_on_hand > 0 THEN
   UPDATE inventory SET quantity = quantity - 1
      WHERE part_number = item_number;
ELSE
   ...
END IF;
```

Relational Operators

The relational operators allow you to compare arbitrarily complex expressions. The following list gives the meaning of each operator:

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. Therefore, to test for *nullity* (the state of being null), do not use the statement

```
IF variable = NULL THEN ...
```

Instead, use the following statement:

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

LIKE Operator

You use the LIKE operator to compare 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.

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

```
ename LIKE 'J%SON'
```

BETWEEN Operator

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

```
45 BETWEEN 38 AND 44
```

IN Operator

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

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

Furthermore, expressions of the form

```
value NOT IN set
```

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

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

Concatenation Operator

Double vertical bars (||) serve as the concatenation operator, which appends one string 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. 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.

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
```

Character Expressions

You can also compare character values for equality or inequality. Comparisons are based on the collating sequence used for the database character set. 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. For example, given the assignments

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

the following expression is true:

```
string1 > string2
```

However, 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.

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

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

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

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

```
100 < tax < 500  -- illegal
```

The debugged version follows:

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

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

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

can be simplified as follows:

```
WHILE NOT done LOOP
   ...
END LOOP;
```

Handling Nulls

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

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.

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

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

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

NOT Operator

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

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

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

Zero-Length Strings

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

```
null_string := TO_VARCHAR2('');
zip_code := SUBSTR(address, 25, 0);
valid := (name != '');
```

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

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

Concatenation Operator

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

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

returns the following value:

```
'applesauce'
```

Functions

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

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

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

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

```
start_date := NVL(hire_date, SYSDATE);
```

The function REPLACE returns the value of its first argument if its second argument is null, whether the optional third argument is present or not. For instance, after the assignment

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

the values of old_string and new_string are the same.

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

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

the value of name is 'goldilocks'

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

Built-In Functions

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

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

Table 2–4 shows the functions in each category. For descriptions of the error-reporting functions, see Chapter 11. For descriptions of the other functions, see *Oracle8i SQL Reference*.

You can use all the functions in SQL statements except the error-reporting functions SQLCODE and SQLERRM. Also, you can use all the functions in procedural statements except the miscellaneous functions DECODE, DUMP, and VSIZE.

The SQL aggregate functions AVG, COUNT, GROUPING, MIN, MAX, SUM, STDDEV, and VARIANCE are not built into PL/SQL. Nevertheless, you can use them in SQL statements (but not in procedural statements).

Table 2–4 Built-in Functions

Error	Number	Character	Conversion	Date	Obj Ref	Misc
SQLCODE	ABS	ASCII	CHARTOROWID	ADD_MONTHS	DEREF	BFILENAME
SQLERRM	ACOS	CHR	CONVERT	LAST_DAY	REF	DECODE
	ASIN	CONCAT	HEXTORAW	MONTHS_BETWEEN	VALUE	DUMP
	ATAN	INITCAP	RAWTOHEX	NEW_TIME		EMPTY_BLOB
	ATAN2	INSTR	ROWIDTOCHAR	NEXT_DAY		EMPTY_CLOB
	CEIL	INSTRB	TO_CHAR	ROUND		GREATEST
	COS	LENGTH	TO_DATE	SYSDATE		LEAST
	COSH	LENGTHB	TO_MULTI_BYTE	TRUNC		NLS_CHARSET_DECL_LEN
	EXP	LOWER	TO_NUMBER			NLS_CHARSET_ID
	FLOOR	LPAD	TO_SINGLE_BYTE			NLS_CHARSET_NAME
	LN	LTRIM				NVL
	LOG	NLS_INITCAP				SYS_CONTEXT
	MOD	NLS_LOWER				SYS_GUID
	POWER	NLSSORT				UID
	ROUND	NLS_UPPER				USER
	SIGN	REPLACE				USERENV
	SIN	RPAD				VSIZE
	SINH	RTRIM				
	SQRT	SOUNDEX				
	TAN	SUBSTR				
	TANH	SUBSTRB				
	TRUNC	TRANSLATE				
		TRIM				
		UPPER				

3

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

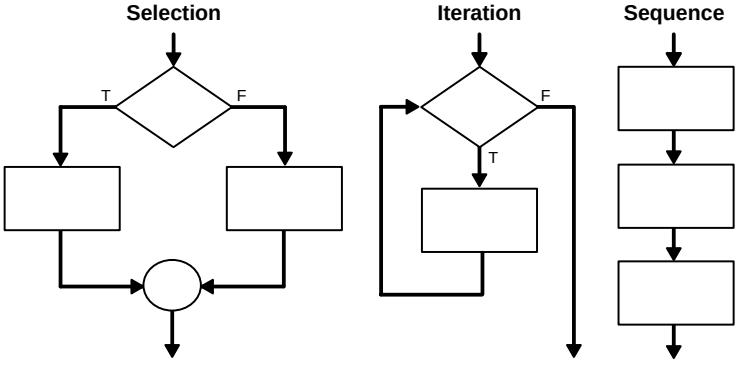
This chapter shows you how to structure the flow of control through a PL/SQL program. You learn how statements are connected by simple but powerful control structures that have a single entry and exit point. Collectively, these structures can handle any situation. Their proper use leads naturally to a well-structured program.

Major Topics

Overview

According to the *structure theorem*, any computer program can be written using the basic control structures shown in Figure 3–1. They can be combined in any way necessary to deal with a given problem.

Figure 3–1 Control Structures



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

Conditional Control: IF Statements

Often, it is necessary to take alternative actions depending on circumstances. The `IF` statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of `IF` statements: `IF-THEN`, `IF-THEN-ELSE`, and `IF-THEN-ELSIF`.

IF-THEN

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

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

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

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

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

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

IF-THEN-ELSE

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

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

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

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

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

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

IF-THEN-ELSIF

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

```
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.

Guidelines

Avoid clumsy IF statements like those in the following example:

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

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

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

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

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

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

```
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 first statement obscures the flow of logic, whereas the second statement reveals it.

Iterative Control: LOOP and EXIT Statements

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

LOOP

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

```
LOOP
   sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

EXIT

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

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

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
   ...
   IF credit_rating < 3 THEN
      ...
      EXIT;  -- illegal
   END IF;
END;
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement. For more information, see "RETURN Statement" on page 7-8.

EXIT-WHEN

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. An example follows:

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

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

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

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

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.

Loop Labels

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<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, you can 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.

WHILE-LOOP

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

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

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

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

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than 25000, the condition is false and the loop is bypassed.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top. Therefore, the sequence of statements is executed at least once. PL/SQL has no such structure, but you can easily build one, as follows:

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

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

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

A statement inside the loop must assign a new value to the Boolean variable. Otherwise, you have an infinite loop. For example, the following LOOP statements are logically equivalent:

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

FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. (Cursor FOR loops, which iterate over the result set of a cursor, are discussed in Chapter 5.) The range is part of an *iteration scheme*, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

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

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

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

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

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

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

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented.

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

Nevertheless, you write the range bounds in ascending (not descending) order.

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

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

Iteration Schemes

The bounds of a loop range can be literals, variables, or expressions but must evaluate to integers. Below are some examples. As you can see, the lower bound need not be 1. However, the loop counter increment (or decrement) must be 1.

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

Some languages provide a STEP clause, which lets you specify a different increment. An example written in BASIC follows:

```
FOR J = 5 TO 15 STEP 5  :REM assign values 5,10,15 to J
   sequence_of_statements  -- J has values 5,10,15
NEXT J
```

PL/SQL has no such structure, but you can easily build one. Consider the following example:

```
FOR j IN 5..15 LOOP  -- assign values 5,6,7,... to j
   IF MOD(j, 5) = 0 THEN  -- pass multiples of 5
      sequence_of_statements  -- j has values 5,10,15
   END IF;
END LOOP;
```

This loop is logically equivalent to the previous BASIC loop. Within the sequence of statements, the loop counter has only the values 5, 10, and 15.

You might prefer the less elegant but more efficient method shown in the example below. Within the sequence of statements, each reference to the loop counter is multiplied by the increment.

```
FOR j IN 1..3 LOOP  -- assign values 1,2,3 to j
   sequence_of_statements  -- each j becomes j*5
END LOOP;
```

Dynamic Ranges

PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

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

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

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

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

Scope Rules

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

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

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

```
DECLARE
   ctr  INTEGER;
BEGIN
   ...
   FOR ctr IN 1..25 LOOP
      ...
      IF ctr > 10 THEN ...  -- refers to loop counter
   END LOOP;
END;
```

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

```
<<main>>
DECLARE
   ctr  INTEGER;
   ...
BEGIN
   ...
   FOR ctr IN 1..25 LOOP
      ...
```

```
      IF main.ctr > 10 THEN  -- refers to global variable
         ...
      END IF;
   END LOOP;
END main;
```

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

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

Using the EXIT Statement

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

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

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

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

Sequential Control: GOTO and NULL Statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in complex, unstructured code (sometimes called *spaghetti code*) that is hard to understand and maintain. So, use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

GOTO Statement

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

```
BEGIN
   ...
   GOTO insert_row;
   ...
   <<insert_row>>
   INSERT INTO emp VALUES ...
END;
```

In the next example, you go to a PL/SQL block farther up in a sequence of statements:

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

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

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

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

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

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

```
DECLARE
   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

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

```
BEGIN
   ...
   GOTO update_row;  -- illegal branch into IF statement
   ...
   IF valid THEN
      ...
      <<update_row>>
      UPDATE emp SET ...
   END IF;
END;
```

Also, a GOTO statement cannot branch from one IF statement clause to another, as the following example shows:

```
BEGIN
   ...
   IF valid THEN
      ...
      GOTO update_row;  -- illegal branch into ELSE clause
   ELSE
      ...
      <<update_row>>
      UPDATE emp SET ...
   END IF;
END;
```

The next example shows that a GOTO statement cannot branch from an enclosing block into a sub-block:

```
BEGIN
   ...
   IF status = 'OBSOLETE' THEN
      GOTO delete_part;  -- illegal branch into sub-block
   END IF;
   ...
```

```
   BEGIN
      ...
      <<delete_part>>
      DELETE FROM parts WHERE ...
   END;
END;
```

Also, a GOTO statement cannot branch out of a subprogram, as the following example shows:

```
DECLARE
   ...
   PROCEDURE compute_bonus (emp_id NUMBER) IS
   BEGIN
      ...
      GOTO update_row;  -- illegal branch out of subprogram
   END;
BEGIN
   ...
   <<update_row>>
   UPDATE emp SET ...
END;
```

Finally, a GOTO statement cannot branch from an exception handler into the current block. For example, the following GOTO statement is illegal:

```
DECLARE
   ...
   pe_ratio  REAL;
BEGIN
   ...
   SELECT price / NVL(earnings, 0) INTO pe_ratio FROM ...
   <<insert_row>>
   INSERT INTO stats VALUES (pe_ratio, ...);
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      pe_ratio := 0;
      GOTO insert_row;  -- illegal branch into current block
END;
```

However, a GOTO statement can branch from an exception handler into an enclosing block.

NULL Statement

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. In a construct allowing alternative actions, the NULL statement serves as a placeholder. It tells readers that the associated alternative has not been overlooked, but that indeed no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

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

Each clause in an IF statement must contain at least one executable statement. The NULL statement is executable, so you can use it in clauses that correspond to circumstances in which no action is taken. In the following example, the NULL statement emphasizes that only top-rated employees get bonuses:

```
IF rating > 90 THEN
   compute_bonus(emp_id);
ELSE
   NULL;
END IF;
```

Also, the NULL statement is a handy way to create stubs when designing applications from the top down. A *stub* is dummy subprogram that allows you to defer the definition of a procedure or function until you test and debug the main program. In the following example, the NULL statement meets the requirement that at least one statement must appear in the executable part of a subprogram:

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

4

Collections and Records

Knowledge is that area of ignorance that we arrange and classify. —Ambrose Bierce

Increasingly, programmers are using collection types such as arrays, bags, lists, nested tables, sets, and trees in traditional database applications. To meet the growing demand, PL/SQL provides the datatypes TABLE and VARRAY, which allow you to declare index-by tables, nested tables and variable-size arrays. In this chapter, you learn how those types let you reference and manipulate collections of data as whole objects. You also learn how the datatype RECORD lets you treat related but dissimilar data as a logical unit.

Major Topics

What Is a Collection?

A *collection* is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers two collection types. Items of type TABLE are either *index-by tables* (Version 2 *PL/SQL tables*) or *nested tables* (which extend the functionality of index-by tables). Items of type VARRAY are *varrays* (short for variable-size arrays).

Collections work like the arrays found in most third-generation programming languages. However, collections can have only one dimension and must be indexed by integers. (In some languages such as Ada and Pascal, arrays can have multiple dimensions and can be indexed by enumeration types.)

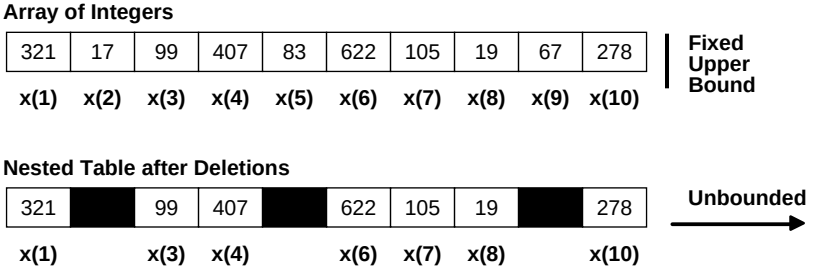
Collections can store instances of an object type and, conversely, can be attributes of an object type. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. Furthermore, you can define collection types in a PL/SQL package, then use them programmatically in your applications.

Understanding Nested Tables

Within the database, nested tables can be considered one-column database tables. Oracle stores the rows of a nested table in no particular order. But, when you retrieve the nested table into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. That gives you array-like access to individual rows.

Within PL/SQL, nested tables are like one-dimensional arrays. However, nested tables differ from arrays in two important ways. First, arrays have a fixed upper bound, but nested tables are unbounded (see Figure 4–1). So, the size of a nested table can increase dynamically.

Figure 4–1 Array versus Nested Table



Second, arrays must be *dense* (have consecutive subscripts). So, you cannot delete individual elements from an array. Initially, nested tables are dense, but they can be *sparse* (have nonconsecutive subscripts). So, you can delete elements from a nested table using the built-in procedure `DELETE`. That might leave gaps in the index, but the built-in function `NEXT` lets you iterate over any series of subscripts.

Nested Tables versus Index-by Tables

Index-by tables and nested tables are similar. For example, they have the same structure, and their individual elements are accessed in the same way (using subscript notation). The main difference is that nested tables can be stored in a database column (hence the term "nested table") but index-by tables cannot.

Nested tables extend the functionality of index-by tables by letting you `SELECT`, `INSERT`, `UPDATE`, and `DELETE` nested tables stored in the database. (Remember, index-by tables cannot be stored in the database.) Also, some collection methods operate only on nested tables and varrays. For example, the built-in procedure `TRIM` cannot be applied to index-by tables.

Another advantage of nested tables is that an uninitialized nested table is atomically null (that is, the table itself is null, not its elements), but an uninitialized index-by table is merely empty. So, you can apply the `IS NULL` comparison operator to nested tables but not to index-by tables.

However, index-by tables also have some advantages. For example, PL/SQL supports implicit (automatic) datatype conversion between host arrays and index-by tables (but not nested tables). So, the most efficient way to pass collections to and from the database server is to use anonymous PL/SQL blocks to bulk-bind input and output host arrays to index-by tables.

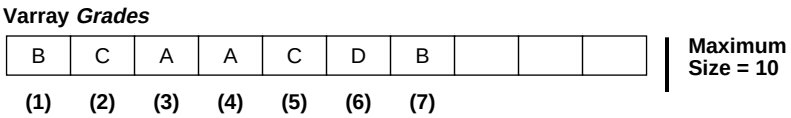
Also, index-by tables are initially sparse. So, they are convenient for storing reference data using a numeric primary key (account numbers or employee numbers for example) as the index.

In some (relatively minor) ways, index-by tables are more flexible than nested tables. For example, subscripts for a nested table are unconstrained. In fact, index-by tables can have negative subscripts (nested tables cannot). Also, some element types are allowed for index-by tables but not for nested tables (see "Referencing Collection Elements" on page 4-10). Finally, to extend a nested table, you must use the built-in procedure `EXTEND`, but to extend an index-by table, you just specify larger subscripts.

Understanding Varrays

Items of type VARRAY are called *varrays*. They allow you to associate a single identifier with an entire collection. This association lets you manipulate the collection as a whole and reference individual elements easily. To reference an element, you use standard subscripting syntax (see Figure 4–2). For example, `Grade(3)` references the third element in varray `Grades`.

Figure 4–2 Varray of Size 10



A varray has a maximum size, which you must specify in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. For example, the current upper bound for varray `Grades` is 7, but you can extend it to 8, 9, or 10. Thus, a varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

Varrays versus Nested Tables

Nested tables differ from varrays in the following ways:

- Varrays have a maximum size, but nested tables do not.
- Varrays are always dense, but nested tables can be sparse. So, you can delete individual elements from a nested table but not from a varray.
- Oracle stores varray data in-line (in the same tablespace). But, Oracle stores nested table data out-of-line in a *store table*, which is a system-generated database table associated with the nested table.
- When stored in the database, varrays retain their ordering and subscripts, but nested tables do not.

Which collection type should you use? That depends on your wants and the size of the collection. A varray is stored as an opaque object, whereas a nested table is stored in a storage table with every element mapped to a row in the table. So, if you want efficient queries, use nested tables. If you want to retrieve entire collections as a whole, use varrays. However, when collections get very large, it becomes impractical to retrieve more than subsets. So, varrays are better suited for small collections.

Defining and Declaring Collections

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

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

and for varrays, use the following syntax:

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

where type_name is a type specifier used later to declare collections, size_limit is a positive integer literal, and element_type is any PL/SQL datatype except

- BINARY_INTEGER, PLS_INTEGER
- BOOLEAN
- BLOB, CLOB (restriction applies only to varrays)
- LONG, LONG RAW
- NATURAL, NATURALN
- NCHAR, NCLOB, NVARCHAR2
- object types with BLOB or CLOB attributes (restriction applies only to varrays)
- object types with TABLE or VARRAY attributes
- POSITIVE, POSITIVEN
- REF CURSOR
- SIGNTYPE
- STRING
- TABLE
- VARRAY

If element_type is a record type, every field in the record must be a scalar type or an object type.

For index-by tables, use the syntax

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

Unlike nested tables and varrays, index-by tables can have the following element types: BINARY_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, PLS_INTEGER, POSITIVE, POSITIVEN, SIGNTYPE, and STRING. That is because nested tables and varrays are intended mainly for database columns. As such, they cannot use PL/SQL-specific types. When declared locally, they could theoretically use those types, but the restriction is preserved for consistency.

Index-by tables are initially sparse. That enables you, for example, to store reference data in a temporary index-by table using a numeric primary key as the index. In the example below, you declare an index-by table of records. Each element of the table stores a row from the emp database table.

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab EmpTabTyp;
BEGIN
   /* Retrieve employee record. */
   SELECT * INTO emp_tab(7468) FROM emp WHERE empno = 7788;
```

When defining a VARRAY type, you must specify its maximum size. In the following example, you define a type that stores up to 366 dates:

```
DECLARE
   TYPE Calendar IS VARRAY(366) OF DATE;
```

To specify the element type, you can use %TYPE, which provides the datatype of a variable or database column. Also, you can use %ROWTYPE, which provides the rowtype of a cursor or database table. Two examples follow:

```
DECLARE
   TYPE EmpList IS TABLE OF emp.ename%TYPE;  -- based on column
   CURSOR c1 IS SELECT * FROM dept;
   TYPE DeptFile IS VARRAY(20) OF c1%ROWTYPE;  -- based on cursor
```

In the next example, you use a RECORD type to specify the element type:

```
DECLARE
   TYPE AnEntry IS RECORD (
      term    VARCHAR2(20),
      meaning VARCHAR2(200));
   TYPE Glossary IS VARRAY(250) OF AnEntry;
```

In the final example, you impose a NOT NULL constraint on the element type:

```
DECLARE
   TYPE EmpList IS TABLE OF emp.empno%TYPE NOT NULL;
```

An initialization clause is not required (or allowed).

Declaring Collections

Once you define a collection type, you can declare collections of that type, as the following SQL*Plus script shows:

```
CREATE TYPE CourseList AS TABLE OF VARCHAR2(10)  -- define type
/
CREATE TYPE Student AS OBJECT (  -- create object
   id_num  INTEGER(4),
   name    VARCHAR2(25),
   address VARCHAR2(35),
   status  CHAR(2),
   courses CourseList)  -- declare nested table as attribute
/
```

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

The script below creates a database column that stores varrays. Each element of the varrays will store a `Project` object.

```
CREATE TYPE Project AS OBJECT(  --create object
   project_no NUMBER(2),
   title      VARCHAR2(35),
   cost       NUMBER(7,2))
/
CREATE TYPE ProjectList AS VARRAY(50) OF Project  -- define VARRAY
type
/
CREATE TABLE department (  -- create database table
   dept_id  NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
   projects ProjectList)  -- declare varray as column
/
```

The following example shows that you can use `%TYPE` to provide the datatype of a previously declared collection:

```
DECLARE
   TYPE Platoon IS VARRAY(20) OF Soldier;
   p1 Platoon;
   p2 p1%TYPE;
```

You can declare collections as the formal parameters of functions and procedures. That way, you can pass collections to stored subprograms and from one subprogram to another. In the following example, you declare a nested table as the formal parameter of a packaged procedure:

```
CREATE PACKAGE personnel AS
   TYPE Staff IS TABLE OF Employee;
   ...
   PROCEDURE award_bonuses (members IN Staff);
END personnel;
```

Also, you can specify a collection type in the RETURN clause of a function specification, as the following example shows:

```
DECLARE
   TYPE SalesForce IS VARRAY(25) OF Salesperson;
   FUNCTION top_performers (n INTEGER) RETURN SalesForce IS ...
```

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

Initializing and Referencing Collections

Until you initialize it, a nested table or varray is atomically null (that is, the collection itself is null, not its elements). To initialize a nested table or varray, you use a *constructor*, which is a system-defined function with the same name as the collection type. This function "constructs" collections from the elements passed to it. In the following example, you pass six elements to constructor `CourseList()`, which returns a nested table containing those elements:

```
DECLARE
   my_courses CourseList;
BEGIN
   my_courses := CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100',
      'PoSc 3141', 'Mktg 3312', 'Engl 2005');
   ...
END;
```

In the next example, you pass three objects to constructor `ProjectList()`, which returns a varray containing those objects:

```
DECLARE
   accounting_projects ProjectList;
BEGIN
   accounting_projects :=
      ProjectList(Project(1, 'Design New Expense Report', 3250),
                  Project(2, 'Outsource Payroll', 12350),
                  Project(3, 'Audit Accounts Payable', 1425));
   ...
END;
```

You need not initialize the whole varray. For example, if a varray has a maximum size of 50, you can pass fewer than 50 elements to its constructor.

Unless you impose the NOT NULL constraint or specify a record type for elements, you can pass null elements to a constructor. An example follows:

```
BEGIN
   my_courses := CourseList('Math 3010', NULL, 'Stat 3202', ...);
```

The next example shows that you can initialize a collection in its declaration, which is a good programming practice:

```
DECLARE
   my_courses CourseList :=
      CourseList('Art 1111', 'Hist 3100', 'Engl 2005', ...);
```

If you call a constructor without arguments, you get an empty but non-null collection, as the following example shows:

```
DECLARE
   TYPE Clientele IS VARRAY(100) OF Customer;
   vips Clientele := Clientele(); -- initialize empty varray
BEGIN
   IF vips IS NOT NULL THEN  -- condition yields TRUE
      ...
   END IF;
END;
```

Except for index-by tables, PL/SQL never calls a constructor implicitly, so you must call it explicitly. Constructor calls are allowed wherever function calls are allowed. That includes the SELECT, VALUES, and SET clauses.

In the example below, you insert a `Student` object into object table `sophomores`. The table constructor `CourseList()` provides a value for attribute `courses`.

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

In the final example, you insert a row into database table `department`. The varray constructor `ProjectList()` provides a value for column `projects`.

```
BEGIN
   INSERT INTO department
      VALUES(60, 'Security', 750400,
         ProjectList(Project(1, 'Issue New Employee Badges', 9500),
                     Project(2, 'Find Missing IC Chips', 2750),
                     Project(3, 'Inspect Emergency Exits', 1900)));
   ...
```

Referencing Collection Elements

Every reference to an element includes a collection name and a subscript enclosed in parentheses. The subscript determines which element is processed. To reference an element, you specify its subscript using the syntax

```
collection_name(subscript)
```

where `subscript` is an expression that yields an integer. For index-by tables, the legal subscript range is -2147483647 .. 2147483647. For nested tables, the legal range is 1 .. 2147483647. And, for varrays, the legal range is 1 .. `size_limit`.

You can reference a collection in all expression contexts. In the following example, you reference an element in nested table `names`:

```
DECLARE
   TYPE Roster IS TABLE OF VARCHAR2(15);
   names Roster := Roster('J Hamil', 'D Caruso', 'R Singh', ...);
   i BINARY_INTEGER;
BEGIN
   ...
   IF names(i) = 'J Hamil' THEN
      ...
   END IF;
END;
```

The next example shows that you can reference the elements of a collection in subprogram calls:

```
DECLARE
   TYPE Roster IS TABLE OF VARCHAR2(15);
   names Roster := Roster('J Hamil', 'D Piro', 'R Singh', ...);
   i BINARY_INTEGER;
BEGIN
   ...
   verify_name(names(i));  -- call procedure
END;
```

When calling a function that returns a collection, use the following syntax to reference elements in the collection:

```
function_name(parameter_list)(subscript)
```

For example, the following call references the third element in the varray returned by function `new_hires`:

```
DECLARE
   TYPE Staff IS VARRAY(20) OF Employee;
   staffer Employee;
   FUNCTION new_hires (hiredate DATE) RETURN Staff IS ...
BEGIN
   staffer := new_hires('16-OCT-96')(3);  -- call function
   ...
END;
```

Assigning and Comparing Collections

One collection can be assigned to another by an `INSERT`, `UPDATE`, `FETCH`, or `SELECT` statement, an assignment statement, or a subprogram call. As the example below shows, the collections must have the same datatype. Having the same element type is not enough.

```
DECLARE
   TYPE Clientele IS VARRAY(100) OF Customer;
   TYPE Vips IS VARRAY(100) OF Customer;
   group1 Clientele := Clientele(...);
   group2 Clientele := Clientele(...);
   group3 Vips := Vips(...);
BEGIN
   group2 := group1;
   group3 := group2;  -- illegal; different datatypes
```

If you assign an atomically null collection to another collection, the other collection becomes atomically null (and must be reinitialized). Consider the following example:

```
DECLARE
   TYPE Clientele IS TABLE OF Customer;
   group1 Clientele := Clientele(...);  -- initialized
   group2 Clientele;  -- atomically null
BEGIN
   IF group1 IS NULL THEN ...  -- condition yields FALSE
   group1 := group2;
   IF group1 IS NULL THEN ...  -- condition yields TRUE
   ...
END;
```

Likewise, if you assign the non-value NULL to a collection, the collection becomes atomically null.

Assigning Collection Elements

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

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

where expression yields a value of the type specified for elements in the collection type definition. If subscript is null or not convertible to an integer, PL/SQL raises the predefined exception VALUE_ERROR. If the collection is atomically null, PL/SQL raises COLLECTION_IS_NULL. Some examples follow:

```
DECLARE
   TYPE NumList IS TABLE OF INTEGER;
   nums NumList := NumList(10,20,30);
   ints NumList;
   ...
BEGIN
   ...
   nums(1) := TRUNC(high/low);
   nums(3) := nums(1);
   nums(2) := ASCII('B');
   /* Assume execution continues despite the raised exception. */
   nums('A') := 40; -- raises VALUE_ERROR
   ints(1) := 15;   -- raises COLLECTION_IS_NULL
END;
```

Comparing Whole Collections

Nested tables and varrays can be atomically null, so they can be tested for nullity, as the following example shows:

```
DECLARE
   TYPE Staff IS TABLE OF Employee;
   members Staff;
BEGIN
   ...
   IF members IS NULL THEN ...  -- condition yields TRUE;
END;
```

However, collections cannot be compared for equality or inequality. For instance, the following `IF` condition is illegal:

```
DECLARE
   TYPE Clientele IS TABLE OF Customer;
   group1 Clientele := Clientele(...);
   group2 Clientele := Clientele(...);
BEGIN
   ...
   IF group1 = group2 THEN  -- causes compilation error
      ...
   END IF;
END;
```

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

Manipulating Collections

Within PL/SQL, collections add flexibility and procedural power. A big advantage is that your program can compute subscripts to process specific elements. A bigger advantage is that the program can use SQL to manipulate in-memory collections.

Some Nested Table Examples

In SQL*Plus, suppose you define object type `Course`, as follows:

```
SQL> CREATE TYPE Course AS OBJECT (
  2  course_no NUMBER(4),
  3  title     VARCHAR2(35),
  4  credits   NUMBER(1));
```

Next, you define TABLE type CourseList, which stores Course objects:

```
SQL> CREATE TYPE CourseList AS TABLE OF Course;
```

Finally, you create database table department, which has a column of type CourseList, as follows:

```
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 because department has a nested table column. The clause identifies the nested table and names a system-generated store table, in which Oracle stores data out-of-line (in another tablespace).

Now, you can populate database table department. In the following example, notice how table constructor CourseList() provides values for column courses:

```
BEGIN
   INSERT INTO department
      VALUES('Psychology', 'Irene Friedman', 'Fulton Hall 133',
         CourseList(Course(1000, 'General Psychology', 5),
                    Course(2100, 'Experimental Psychology', 4),
                    Course(2200, 'Psychological Tests', 3),
                    Course(2250, 'Behavior Modification', 4),
                    Course(3540, 'Groups and Organizations', 3),
                    Course(3552, 'Human Factors in Busines', 4),
                    Course(4210, 'Theories of Learning', 4),
                    Course(4320, 'Cognitive Processes', 4),
                    Course(4410, 'Abnormal Psychology', 4)));
   INSERT INTO department
      VALUES('History', 'John Whalen', 'Applegate Hall 142',
         CourseList(Course(1011, 'History of Europe I', 4),
                    Course(1012, 'History of Europe II', 4),
                    Course(1202, 'American History', 5),
                    Course(2130, 'The Renaissance', 3),
                    Course(2132, 'The Reformation', 3),
                    Course(3105, 'History of Ancient Greece', 4),
                    Course(3321, 'Early Japan', 4),
                    Course(3601, 'Latin America Since 1825', 4),
                    Course(3702, 'Medieval Islamic History', 4)));
```

```
   INSERT INTO department
      VALUES('English', 'Lynn Saunders', 'Breakstone Hall 205',
         CourseList(Course(1002, 'Expository Writing', 3),
                    Course(2020, 'Film and Literature', 4),
                    Course(2418, 'Modern Science Fiction', 3),
                    Course(2810, 'Discursive Writing', 4),
                    Course(3010, 'Modern English Grammar', 3),
                    Course(3720, 'Introduction to Shakespeare', 4),
                    Course(3760, 'Modern Drama', 4),
                    Course(3822, 'The Short Story', 4),
                    Course(3870, 'The American Novel', 5)));
END;
```

In the following example, you revise the list of courses offered by the English Department:

```
DECLARE
   new_courses CourseList :=
      CourseList(Course(1002, 'Expository Writing', 3),
                 Course(2020, 'Film and Literature', 4),
                 Course(2810, 'Discursive Writing', 4),
                 Course(3010, 'Modern English Grammar', 3),
                 Course(3550, 'Realism and Naturalism', 4),
                 Course(3720, 'Introduction to Shakespeare', 4),
                 Course(3760, 'Modern Drama', 4),
                 Course(3822, 'The Short Story', 4),
                 Course(3870, 'The American Novel', 4),
                 Course(4210, '20th-Century Poetry', 4),
                 Course(4725, 'Advanced Workshop in Poetry', 5));
BEGIN
   UPDATE department
      SET courses = new_courses WHERE name = 'English';
END;
```

In the next example, you retrieve all the courses offered by the Psychology Department into a local nested table:

```
DECLARE
   psyc_courses CourseList;
BEGIN
   SELECT courses INTO psyc_courses FROM department
      WHERE name = 'Psychology';
   ...
END;
```

Some Varray Examples

In SQL*Plus, suppose you define object type `Project`, as follows:

```
SQL> CREATE TYPE Project AS OBJECT (
  2  project_no NUMBER(2),
  3  title      VARCHAR2(35),
  4  cost       NUMBER(7,2));
```

Next, you define VARRAY type `ProjectList`, which stores `Project` objects:

```
SQL> CREATE TYPE ProjectList AS VARRAY(50) OF Project;
```

Finally, you create relational table `department`, which has a column of type `ProjectList`, as follows:

```
SQL> CREATE TABLE department (
  2  dept_id  NUMBER(2),
  3  name     VARCHAR2(15),
  4  budget   NUMBER(11,2),
  5  projects ProjectList);
```

Each item in column `projects` is a varray that will store the projects scheduled for a given department.

Now, you are ready to populate relational table `department`. In the following example, notice how varray constructor `ProjectList()` provides values for column `projects`:

```
BEGIN
   INSERT INTO department
      VALUES(30, 'Accounting', 1205700,
         ProjectList(Project(1, 'Design New Expense Report', 3250),
                     Project(2, 'Outsource Payroll', 12350),
                     Project(3, 'Evaluate Merger Proposal', 2750),
                     Project(4, 'Audit Accounts Payable', 1425)));
   INSERT INTO department
      VALUES(50, 'Maintenance', 925300,
         ProjectList(Project(1, 'Repair Leak in Roof', 2850),
                     Project(2, 'Install New Door Locks', 1700),
                     Project(3, 'Wash Front Windows', 975),
                     Project(4, 'Repair Faulty Wiring', 1350),
                     Project(5, 'Winterize Cooling System', 1125)));
```

```
   INSERT INTO department
      VALUES(60, 'Security', 750400,
         ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                     Project(2, 'Find Missing IC Chips', 2750),
                     Project(3, 'Upgrade Alarm System', 3350),
                     Project(4, 'Inspect Emergency Exits', 1900)));
END;
```

In the following example, you update the list of projects assigned to the Security Department:

```
DECLARE
   new_projects ProjectList :=
      ProjectList(Project(1, 'Issue New Employee Badges', 13500),
                  Project(2, 'Develop New Patrol Plan', 1250),
                  Project(3, 'Inspect Emergency Exits', 1900),
                  Project(4, 'Upgrade Alarm System', 3350),
                  Project(5, 'Analyze Local Crime Stats', 825));
BEGIN
   UPDATE department
      SET projects = new_projects WHERE dept_id = 60;
END;
```

In the next example, you retrieve all the projects for the Accounting Department into a local varray:

```
DECLARE
   my_projects ProjectList;
BEGIN
   SELECT projects INTO my_projects FROM department
      WHERE dept_id = 30;
   ...
END;
```

In the final example, you delete the Accounting Department and its project list from table `department`:

```
BEGIN
   DELETE FROM department WHERE dept_id = 30;
END;
```

Manipulating Individual Elements

So far, you have manipulated whole collections. Within SQL, to manipulate the individual elements of a collection, use the operator TABLE. The operand of TABLE is a subquery that returns a single column value for you to manipulate. That value is a nested table or varray.

In the following example, you add a row to the History Department nested table stored in column courses:

```
BEGIN
   INSERT INTO
      TABLE(SELECT courses FROM department WHERE name = 'History')
      VALUES(3340, 'Modern China', 4);
END;
```

In the next example, you revise the number of credits for two courses offered by the Psychology Department:

```
DECLARE
   adjustment INTEGER DEFAULT 1;
BEGIN
   UPDATE TABLE(SELECT courses FROM department
         WHERE name = 'Psychology')
      SET credits = credits + adjustment
      WHERE course_no IN (2200, 3540);
END;
```

In the following example, you retrieve the number and title of a specific course offered by the History Department:

```
DECLARE
   my_course_no NUMBER(4);
   my_title VARCHAR2(35);
BEGIN
   SELECT course_no, title INTO my_course_no, my_title
      FROM TABLE(SELECT courses FROM department
         WHERE name = 'History')
      WHERE course_no = 3105;
   ...
END;
```

In the next example, you delete all 5-credit courses offered by the English Department:

```
BEGIN
   DELETE TABLE(SELECT courses FROM department
         WHERE name = 'English')
      WHERE credits = 5;
END;
```

In the following example, you retrieve the title and cost of the Maintenance Department's fourth project from the varray column `projects`:

```
DECLARE
   my_cost  NUMBER(7,2);
   my_title VARCHAR2(35);
BEGIN
   SELECT cost, title INTO my_cost, my_title
      FROM TABLE(SELECT projects FROM department
         WHERE dept_id = 50)
      WHERE project_no = 4;
   ...
END;
```

Currently, you cannot reference the individual elements of a varray in an `INSERT`, `UPDATE`, or `DELETE` statement. So, you must use PL/SQL procedural statements. In the following example, stored procedure `add_project` inserts a new project into a department's project list at a given position:

```
CREATE PROCEDURE add_project (
   dept_no     IN NUMBER,
   new_project IN Project,
   position    IN NUMBER) AS
   my_projects ProjectList;
BEGIN
   SELECT projects INTO my_projects FROM department
      WHERE dept_no = dept_id FOR UPDATE OF projects;
   my_projects.EXTEND;  -- make room for new project
   /* Move varray elements forward. */
   FOR i IN REVERSE position..my_projects.LAST - 1 LOOP
      my_projects(i + 1) := my_projects(i);
   END LOOP;
   my_projects(position) := new_project;  -- add new project
   UPDATE department SET projects = my_projects
      WHERE dept_no = dept_id;
END add_project;
```

The following stored procedure updates a given project:

```
CREATE PROCEDURE update_project (
   dept_no   IN NUMBER,
   proj_no   IN NUMBER,
   new_title IN VARCHAR2 DEFAULT NULL,
   new_cost  IN NUMBER DEFAULT NULL) AS
   my_projects ProjectList;
BEGIN
   SELECT projects INTO my_projects FROM department
      WHERE dept_no = dept_id FOR UPDATE OF projects;
   /* Find project, update it, then exit loop immediately. */
   FOR i IN my_projects.FIRST..my_projects.LAST LOOP
      IF my_projects(i).project_no = proj_no THEN
         IF new_title IS NOT NULL THEN
            my_projects(i).title := new_title;
         END IF;
         IF new_cost IS NOT NULL THEN
            my_projects(i).cost := new_cost;
         END IF;
         EXIT;
      END IF;
   END LOOP;
   UPDATE department SET projects = my_projects
      WHERE dept_no = dept_id;
END update_project;
```

Manipulating Local Collections

Within PL/SQL, to manipulate a local collection, use the operators TABLE and CAST. The operands of CAST are a collection declared locally (in a PL/SQL anonymous block for example) and a SQL collection type. CAST converts the local collection to the specified type. That way, you can manipulate the collection as if it were a SQL database table. In the following example, you count the number of differences between a revised course list and the original (notice that the number of credits for course 3720 changed from 4 to 3):

```
DECLARE
   revised CourseList :=
      CourseList(Course(1002, 'Expository Writing',        3),
                 Course(2020, 'Film and Literature',       4),
                 Course(2810, 'Discursive Writing',        4),
                 Course(3010, 'Modern English Grammar ',   3),
                 Course(3550, 'Realism and Naturalism',    4),
```

```
                Course(3720, 'Introduction to Shakespeare',3),
                Course(3760, 'Modern Drama',               4),
                Course(3822, 'The Short Story',            4),
                Course(3870, 'The American Novel',         5),
                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)) AS 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 Collection Methods

The following collection methods help generalize code, 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. The syntax follows:

```
collection_name.method_name[(parameters)]
```

Collection methods can be called from procedural statements but not from SQL statements. EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, and NEXT are functions, which appear as part of an expression. EXTEND, TRIM, and DELETE are procedures, which appear as a statement. Also, EXISTS, PRIOR, NEXT, TRIM, EXTEND, and DELETE take parameters. Each parameter must be an integer expression.

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

Using 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. In the following example, PL/SQL executes the assignment statement only if element i exists:

```
IF courses.EXISTS(i) THEN courses(i) := new_course; END IF;
```

When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

Using COUNT

COUNT returns the number of elements that a collection currently contains. For instance, if varray projects contains 15 elements, the following IF condition is true:

```
IF projects.COUNT = 15 THEN ...
```

COUNT is useful because the current size of a collection is not always known. For example, if you fetch a column of Oracle data into a nested table, how many elements does the table contain? COUNT gives you the answer.

You can use COUNT wherever an integer expression is allowed. In the next example, you use COUNT to specify the upper bound of a loop range:

```
FOR i IN 1..courses.COUNT LOOP ...
```

For varrays, COUNT always equals LAST. For nested tables, COUNT normally equals LAST. But, if you delete elements from the middle of a nested table, COUNT becomes smaller than LAST.

When tallying elements, COUNT ignores deleted elements.

Using 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). For instance, if the maximum size of varray projects is 25 elements, the following IF condition is true:

```
IF projects.LIMIT = 25 THEN ...
```

You can use LIMIT wherever an integer expression is allowed. In the following example, you use LIMIT to determine if you can add 20 more elements to varray projects:

```
IF (projects.COUNT + 20) < projects.LIMIT THEN ...
```

Using FIRST and LAST

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection. If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same index number, as the following example shows:

```
IF courses.FIRST = courses.LAST THEN ...  -- only one element
```

The next example shows that you can use FIRST and LAST to specify the lower and upper bounds of a loop range provided each element in that range exists:

```
FOR i IN courses.FIRST..courses.LAST LOOP ...
```

In fact, you can use FIRST or LAST wherever an integer expression is allowed. In the following example, you use FIRST to initialize a loop counter:

```
i := courses.FIRST;
WHILE i IS NOT NULL LOOP ...
```

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

When scanning elements, FIRST and LAST ignore deleted elements.

Using PRIOR and NEXT

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

PRIOR and NEXT do not wrap from one end of a collection to the other. For example, the following statement assigns NULL to n because the first element in a collection has no predecessor:

```
n := courses.PRIOR(courses.FIRST);  -- assigns NULL to n
```

PRIOR is the inverse of NEXT. For instance, if element i exists, the following statement assigns element i to itself:

```
projects(i) := projects.PRIOR(projects.NEXT(i));
```

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

```
i := courses.FIRST;  -- get subscript of first element
WHILE i IS NOT NULL LOOP
   -- do something with courses(i)
   i := courses.NEXT(i);  -- get subscript of next element
END LOOP;
```

When traversing elements, PRIOR and NEXT ignore deleted elements.

Using EXTEND

To increase the size of a collection, use EXTEND. This procedure has three forms. EXTEND appends one null element to a collection. EXTEND(n) appends n null elements to a collection. EXTEND(n,i) appends n copies of the *i*th element to a collection. For example, the following statement appends 5 copies of element 1 to nested table courses:

```
courses.EXTEND(5,1);
```

You cannot use EXTEND to initialize an atomically null collection. Also, if you impose the NOT NULL constraint on a TABLE or VARRAY type, you cannot apply the first two forms of EXTEND to collections of that type.

EXTEND operates on the internal size of a collection, which includes any deleted elements. So, if EXTEND encounters deleted elements, it includes them in its tally. PL/SQL keeps placeholders for deleted elements so that you can replace them if you wish. Consider the following example:

```
DECLARE
   TYPE CourseList IS TABLE OF VARCHAR2(10);
   courses CourseList;
BEGIN
   courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');
   courses.DELETE(3);  -- delete element 3
   /* PL/SQL keeps a placeholder for element 3. So, the
      next statement appends element 4, not element 3. */
```

```
courses.EXTEND;  -- append one null element
/* Now element 4 exists, so the next statement does
   not raise SUBSCRIPT_BEYOND_COUNT. */
courses(4) := 'Engl 2005';
```

When it includes deleted elements, the internal size of a nested table differs from the values returned by COUNT and LAST. For instance, if you initialize a nested table with five elements, then delete elements 2 and 5, the internal size is 5, COUNT returns 3, and LAST returns 4. All deleted elements (whether leading, in the middle, or trailing) are treated alike.

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

```
courses.TRIM(3);
```

If n is greater than COUNT, TRIM(n) raises SUBSCRIPT_BEYOND_COUNT.

TRIM operates on the internal size of a collection. So, 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'
```

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.

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

Using DELETE

This procedure has three forms. DELETE removes all elements from a collection. DELETE(n) removes the *n*th element from a nested table. If n is null, DELETE(n) does nothing. DELETE(m,n) removes all elements in the range m..n from an index-by table or a nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing. Some examples follow:

```
BEGIN
   ...
   courses.DELETE(2);    -- deletes element 2
   courses.DELETE(7,7);  -- deletes element 7
   courses.DELETE(6,3);  -- does nothing
   courses.DELETE(3,6);  -- deletes elements 3 through 6
   projects.DELETE;      -- deletes all elements
END;
```

Varrays are dense, so you cannot delete their individual elements.

If an element to be deleted does not exist, DELETE simply skips it; no exception is raised. PL/SQL keeps placeholders for deleted elements. So, you can replace a deleted element simply by assigning it a new value.

DELETE allows you to maintain sparse nested tables. In the following example, you retrieve nested table prospects into a temporary table, prune it, then store it back in the database:

```
DECLARE
   my_prospects ProspectList;
   revenue      NUMBER;
BEGIN
   SELECT prospects INTO my_prospects FROM customers WHERE ...
   FOR i IN my_prospects.FIRST..my_prospects.LAST LOOP
      estimate_revenue(my_prospects(i), revenue);  -- call procedure
      IF revenue < 25000 THEN
         my_prospects.DELETE(i);
      END IF;
   END LOOP;
   UPDATE customers SET prospects = my_prospects WHERE ...
```

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

Applying Methods to Collection Parameters

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. So, you can apply the built-in collection methods (FIRST, LAST, COUNT, and so on) to such parameters. In the following example, a nested table is declared as the formal parameter of a packaged procedure:

```
CREATE PACKAGE personnel AS
   TYPE Staff IS TABLE OF Employee;
   ...
   PROCEDURE award_bonuses (members IN Staff);
END personnel;
CREATE PACKAGE BODY personnel AS
   ...
   PROCEDURE award_bonuses (members IN Staff) IS
   BEGIN
      ...
      IF members.COUNT > 10 THEN  -- apply method
         ...
      END IF;
   END;
END personnel;
```

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 table
   nums(NULL) := 3          -- raises VALUE_ERROR              (2)
   nums(0) := 3;            -- raises SUBSCRIPT_OUTSIDE_LIMIT  (3)
   nums(3) := 3;            -- raises SUBSCRIPT_BEYOND_COUNT   (4)
   nums.DELETE(1);  -- delete element 1
   IF nums(1) = 1 THEN ...  -- raises NO_DATA_FOUND            (5)
```

In the first case, the nested table is atomically null. In the second case, the subscript is null. In the third case, the subscript is outside the legal range. In the fourth case, the subscript exceeds the number of elements in the table. In the fifth case, the subscript designates a deleted element.

The following list shows when a given exception is raised:

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
SUBSCRIPT_BEYOND_COUNT	a subscript exceeds the number of elements in a collection
SUBSCRIPT_OUTSIDE_LIMIT	a subscript is outside the legal range
VALUE_ERROR	a subscript is null or not convertible to an integer

In some cases, you can pass "invalid" subscripts to a method without raising an exception. For instance, when you pass a null subscript to procedure DELETE, it does nothing. Also, you can replace deleted elements without raising NO_DATA_FOUND, as the following example shows:

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

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

```
CREATE PACKAGE pkg1 AS
   TYPE NumList IS VARRAY(25) OF NUMBER(4);
   PROCEDURE delete_emps (emp_list NumList);
   ...
END pkg1;
```

```
CREATE PACKAGE BODY pkg1 AS
   PROCEDURE delete_emps (emp_list NumList) IS ...
   ...
END pkg1;
```

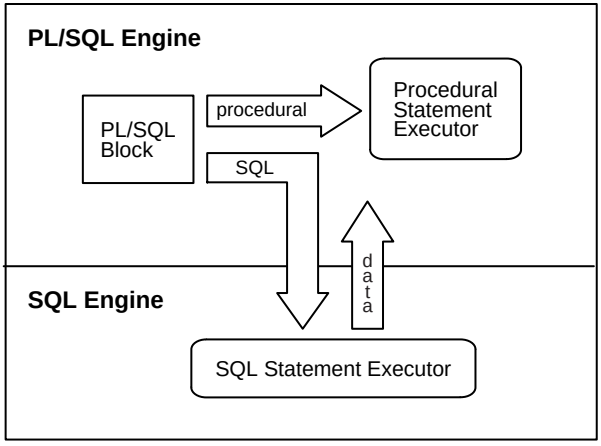
When you run the PL/SQL block below, the second procedure call fails with a *wrong number or types of arguments* error. That is because the packaged and local VARRAY types are incompatible even though their definitions are identical.

```
DECLARE
   TYPE NumList IS VARRAY(25) OF NUMBER(4);
   emps  pkg1.NumList := pkg1.NumList(7369, 7499);
   emps2 NumList := NumList(7521, 7566);
BEGIN
   pkg1.delete_emps(emps);
   pkg1.delete_emps(emps2);  -- causes a compilation error
END;
```

Taking Advantage of Bulk Binds

Embedded in the Oracle RDBMS, the PL/SQL engine accepts any valid PL/SQL block or subprogram. As Figure 4–3 shows, the PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine, which executes the SQL statements and, in some cases, returns data to the PL/SQL engine.

Figure 4–3 Context Switching



Each context switch between the PL/SQL and SQL engines adds to overhead. So, if many switches are required, performance suffers. That can happen when SQL statements execute inside a loop using collection (index-by table, nested table, varray, or host array) elements as bind variables. For example, the following DELETE statement is sent to the SQL engine with each iteration of the FOR loop:

```
DECLARE
   TYPE NumList IS VARRAY(20) OF NUMBER;
   depts NumList := NumList(10, 30, 70, ...);  -- department numbers
BEGIN
   ...
   FOR i IN depts.FIRST..depts.LAST LOOP
      ...
      DELETE FROM emp WHERE deptno = depts(i);
   END LOOP;
END;
```

In such cases, if the SQL statement affects five or more database rows, the use of bulk binds can improve performance considerably.

How Do Bulk Binds Improve Performance?

The assigning of values to PL/SQL variables in SQL statements is called *binding*. The binding of an entire collection at once is called *bulk binding*. Bulk binds improve performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth. For example, the following DELETE statement is sent to the SQL engine just once, with an entire nested table:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   mgrs NumList := NumList(7566, 7782, ...);  -- manager numbers
BEGIN
   ...
   FORALL i IN mgrs.FIRST..mgrs.LAST
      DELETE FROM emp WHERE mgr = mgrs(i);
END;
```

In the example below, 5000 part numbers and names are loaded into index-by tables. Then, all table elements are inserted into a database table twice. First, they are inserted using a FOR loop, which completes in 38 seconds. Then, they are bulk-inserted using a FORALL statement, which completes in only 3 seconds.

```
SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE parts (pnum NUMBER(4), pname CHAR(15));

Table created.

SQL> GET test.sql
  1  DECLARE
  2     TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY_INTEGER;
  3     TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
  4     pnums  NumTab;
  5     pnames NameTab;
  6     t1 CHAR(5);
  7     t2 CHAR(5);
  8     t3 CHAR(5);
  9     PROCEDURE get_time (t OUT NUMBER) IS
10     BEGIN SELECT TO_CHAR(SYSDATE,'SSSSS') INTO t FROM dual; END;
11  BEGIN
12     FOR j IN 1..5000 LOOP  -- load index-by tables
13        pnums(j) := j;
14        pnames(j) := 'Part No. ' || TO_CHAR(j); 15     END LOOP;
16     get_time(t1);
17     FOR i IN 1..5000 LOOP  -- use FOR loop
18        INSERT INTO parts VALUES (pnums(i), pnames(i));
19     END LOOP;
20     get_time(t2);
21     FORALL i IN 1..5000  -- use FORALL statement
22        INSERT INTO parts VALUES (pnums(i), pnames(i));
23     get_time(t3);
24     DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
25     DBMS_OUTPUT.PUT_LINE('---------------------');
26     DBMS_OUTPUT.PUT_LINE('FOR loop: ' || TO_CHAR(t2 - t1));
27     DBMS_OUTPUT.PUT_LINE('FORALL:   ' || TO_CHAR(t3 - t2));
28* END;
SQL> /
Execution Time (secs)
---------------------
FOR loop: 38
FORALL:   3

PL/SQL procedure successfully completed.
```

To bulk-bind input collections, use the FORALL statement. To bulk-bind output collections, use the BULK COLLECT clause.

Using the FORALL Statement

The keyword FORALL instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is *not* a FOR loop. Its syntax follows:

```
FORALL index IN lower_bound..upper_bound
   sql_statement;
```

The index can be referenced only within the FORALL statement and only as a collection subscript. The SQL statement must be an INSERT, UPDATE, or DELETE statement that references collection elements. And, the bounds must specify a valid range of consecutive index numbers. The SQL engine executes the SQL statement once for each index number in the range. As the following example shows, you can use the bounds to bulk-bind arbitrary slices of a collection:

```
DECLARE
   TYPE NumList IS VARRAY(15) OF NUMBER;
   depts NumList := NumList();
BEGIN
   -- fill varray here
   ...
   FORALL j IN 6..10  -- bulk-bind middle third of varray
      UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
END;
```

The SQL statement can reference more than one collection. However, the PL/SQL engine bulk-binds only subscripted collections. So, in the following example, it does not bulk-bind the collection sals, which is passed to the function median:

```
FORALL i IN 1..20
   INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);
```

The next example shows that the collection subscript cannot be an expression:

```
FORALL j IN mgrs.FIRST..mgrs.LAST
   DELETE FROM emp WHERE mgr = mgrs(j+1);  -- illegal subscript
```

All collection elements in the specified range must exist. If an element is missing or was deleted, you get an error, as the following example shows:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(10, 20, 30, 40);
```

```
BEGIN
   depts.DELETE(3);  -- delete third element
   FORALL i IN depts.FIRST..depts.LAST
      DELETE FROM emp WHERE deptno = depts(i);
      -- raises an "element does not exist" exception
END;
```

Rollback Behavior

If a FORALL statement fails, database changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous executions are *not* rolled back. For example, suppose you create a database table that stores department numbers and job titles, as follows:

```
CREATE TABLE emp2 (deptno NUMBER(2), job VARCHAR2(15));
```

Next, you insert some rows into the table, as follows:

```
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(20, 'Bookkeeper');  -- 10-char job title
INSERT INTO emp2 VALUES(30, 'Analyst');
INSERT INTO emp2 VALUES(30, 'Analyst');
```

Then, you try to append the 7-character string `' (temp)'` to certain job titles using the following UPDATE statement:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(10, 20);
BEGIN
   FORALL j IN depts.FIRST..depts.LAST
      UPDATE emp2 SET job = job || ' (temp)'
         WHERE deptno = depts(j);
         -- raises a "value too large" exception
EXCEPTION
   WHEN OTHERS THEN
      COMMIT;
END;
```

The SQL engine executes the UPDATE statement twice, once for each index number in the specified range; that is, once for `depts(10)` and once for `depts(20)`. The first execution succeeds, but the second execution fails because the string value `'Bookkeeper (temp)'` is too large for the `job` column. In this case, only the second execution is rolled back.

Using the BULK COLLECT Clause

The keywords BULK COLLECT tell the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. You can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses. Here is the syntax:

```
... BULK COLLECT INTO collection_name[, collection_name] ...
```

The SQL engine bulk-binds all collections referenced in the INTO list. The corresponding columns must store scalar (not composite) values. In the following example, the SQL engine loads the entire empno and ename database columns into nested tables before returning the tables to the PL/SQL engine:

```
DECLARE
   TYPE NumTab IS TABLE OF emp.empno%TYPE;
   TYPE NameTab IS TABLE OF emp.ename%TYPE;
   enums NumTab;  -- no need to initialize
   names NameTab;
BEGIN
   SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;
   ...
END;
```

The SQL engine initializes and extends collections for you. (However, it cannot extend varrays beyond their maximum size.) Then, starting at index 1, it inserts elements consecutively and overwrites any pre-existent elements.

The SQL engine bulk-binds entire database columns. So, if a table has 50,000 rows, the engine loads 50,000 column values into the target collection. However, you can use the pseudocolumn ROWNUM to limit the number of rows processed. In the following example, you limit the number of rows to 100:

```
DECLARE
   TYPE NumTab IS TABLE OF emp.empno%TYPE;
   sals NumTab;
BEGIN
   SELECT sal BULK COLLECT INTO sals FROM emp WHERE ROWNUM <= 100;
   ...
END;
```

Bulk Fetching

The following example shows that you can bulk-fetch from a cursor into one or more collections:

```
DECLARE
   TYPE NameTab IS TABLE OF emp.ename%TYPE;
   TYPE SalTab IS TABLE OF emp.sal%TYPE;
   names NameTab;
   sals  SalTab;
   CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO names, sals;
   ...
END;
```

Restriction: You cannot bulk-fetch from a cursor into a collection of records, as the following example shows:

```
DECLARE
   TYPE EmpRecTab IS TABLE OF emp%ROWTYPE;
   emp_recs EmpRecTab;
   CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO emp_recs;  -- illegal
   ...
END;
```

Using FORALL and BULK COLLECT Together

You can combine the BULK COLLECT clause with a FORALL statement, in which case, the SQL engine bulk-binds column values incrementally. In the following example, if collection depts has 3 elements, each of which causes 5 rows to be deleted, then collection enums has 15 elements when the statement completes:

```
FORALL j IN depts.FIRST..depts.LAST
   DELETE FROM emp WHERE empno = depts(j)
      RETURNING empno BULK COLLECT INTO enums;
```

The column values returned by each execution are added to the values returned previously. (With a FOR loop, the previous values are overwritten.)

Restriction: You cannot use the SELECT ... BULK COLLECT statement in a FORALL statement.

Using Host Arrays

Client-side programs can use anonymous PL/SQL blocks to bulk-bind input and output host arrays. In fact, that is the most efficient way to pass collections to and from the database server.

Host arrays are declared in a host environment such as an OCI or Pro*C program and must be prefixed with a colon to distinguish them from PL/SQL collections. In the example below, an input host array is used in a DELETE statement. At run time, the anonymous PL/SQL block is sent to the database server for execution.

```
DECLARE
   ...
BEGIN
   -- assume that values were assigned to the host array
   -- and host variables in the host environment
   FORALL i IN :lower..:upper
      DELETE FROM emp WHERE deptno = :depts(i);
   ...
END;
```

You cannot use collection methods such as FIRST and LAST with host arrays. For example, the following statement is illegal:

```
FORALL i IN :depts.FIRST..:depts.LAST  -- illegal
   DELETE FROM emp WHERE deptno = :depts(i);
```

Using Cursor Attributes

To process SQL data manipulation statements, the SQL engine opens an implicit cursor named SQL. This cursor's attributes (%FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT) return useful information about the most recently executed SQL data manipulation statement.

The SQL cursor has only one composite attribute, %BULK_ROWCOUNT, which has the semantics of an index-by table. Its *i*th element stores the number of rows processed by the *i*th execution of a SQL statement. If the *i*th execution affects no rows, %BULK_ROWCOUNT(i) returns zero. An example follows:

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(10, 20, 50);
```

```
BEGIN
   FORALL j IN depts.FIRST..depts.LAST
      UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
   IF SQL%BULK_ROWCOUNT(3) = 0 THEN
      ...
   END IF
END;
```

%BULK_ROWCOUNT and the FORALL statement use the same subscripts. For instance, if the FORALL statement uses the range -5..10, so does %BULK_ROWCOUNT.

Note: Only index-by tables can have negative subscripts.

You can also use the scalar attributes %FOUND, %NOTFOUND, and %ROWCOUNT with bulk binds. For example, %ROWCOUNT returns the total number of rows processed by all executions of the SQL statement.

%FOUND and %NOTFOUND refer only to the last execution of the SQL statement. However, you can use %BULK_ROWCOUNT to infer their values for individual executions. For example, when %BULK_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

Restrictions: %BULK_ROWCOUNT cannot be assigned to other collections. Also, it cannot be passed as a parameter to subprograms.

What Is a Record?

A *record* is a group of related data items stored in *fields*, each with its own name and datatype. Suppose you have various data about an employee such as name, salary, and hire date. These items are logically related but dissimilar in type. A record containing a field for each item lets you treat the data as a logical unit. Thus, records make it easier to organize and represent information.

The attribute %ROWTYPE lets you declare a record that represents a row in a database table. However, you cannot specify the datatypes of fields in the record or declare fields of your own. The datatype RECORD lifts those restrictions and lets you define your own records.

Defining and Declaring Records

To create records, you define a RECORD type, then declare records of that type. You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package using the syntax

```
TYPE type_name IS RECORD (field_declaration[,field_declaration]...);
```

where field_declaration stands for

```
field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
```

and where type_name is a type specifier used later to declare records, field_type is any PL/SQL datatype except REF CURSOR, and expression yields a value of the same type as field_type.

Note: Unlike TABLE and VARRAY types, RECORD types cannot be CREATEd and stored in the database.

You can use %TYPE and %ROWTYPE to specify field types. In the following example, you define a RECORD type named DeptRec:

```
DECLARE
   TYPE DeptRec IS RECORD (
      dept_id   dept.deptno%TYPE,
      dept_name VARCHAR2(15),
      dept_loc  VARCHAR2(15));
```

Notice that field declarations are like variable declarations. Each field has a unique name and specific datatype. So, the value of a record is actually a collection of values, each of some simpler type.

As the example below shows, PL/SQL lets you define records that contain objects, collections, and other records (called *nested* records). However, object types cannot have attributes of type RECORD.

```
DECLARE
   TYPE TimeRec IS RECORD (
      seconds SMALLINT,
      minutes SMALLINT,
      hours   SMALLINT);
   TYPE FlightRec IS RECORD (
      flight_no   INTEGER,
      plane_id     VARCHAR2(10),
      captain      Employee,  -- declare object
```

```
   passengers   PassengerList,  -- declare varray
   depart_time  TimeRec,  -- declare nested record
   airport_code VARCHAR2(10));
```

The next example shows that you can specify a RECORD type in the RETURN clause of a function specification. That allows the function to return a user-defined record of the same type.

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_id    INTEGER
      last_name VARCHAR2(15),
      dept_num  INTEGER(2),
      job_title VARCHAR2(15),
      salary    REAL(7,2));
   ...
   FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRec IS ...
```

Declaring Records

Once you define a RECORD type, you can declare records of that type, as the following example shows:

```
DECLARE
   TYPE StockItem IS RECORD (
      item_no     INTEGER(3),
      description VARCHAR2(50),
      quantity    INTEGER,
      price       REAL(7,2));
   item_info StckItem;  -- declare record
```

The identifier item_info represents an entire record.

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions. An example follows:

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_id    emp.empno%TYPE,
      last_name VARCHAR2(10),
      job_title VARCHAR2(15),
      salary    NUMBER(7,2));
   ...
   PROCEDURE raise_salary (emp_info EmpRec);
```

Initializing and Referencing Records

The example below shows that you can initialize a record in its type definition. When you declare a record of type TimeRec, its three fields assume an initial value of zero.

```
DECLARE
   TYPE TimeRec IS RECORD (
      secs SMALLINT := 0,
      mins SMALLINT := 0,
      hrs  SMALLINT := 0);
```

The next example shows that you can impose the NOT NULL constraint on any field, and so prevent the assigning of nulls to that field. Fields declared as NOT NULL must be initialized.

```
DECLARE
   TYPE StockItem IS RECORD (
      item_no     INTEGER(3) NOT NULL := 999,
      description VARCHAR2(50),
      quantity    INTEGER,
      price       REAL(7,2));
```

Referencing Records

Unlike elements in a collection, which are accessed using subscripts, fields in a record are accessed by name. To reference an individual field, use dot notation and the following syntax:

```
record_name.field_name
```

For example, you reference field hire_date in record emp_info as follows:

```
emp_info.hire_date ...
```

When calling a function that returns a user-defined record, use the following syntax to reference fields in the record:

```
function_name(parameter_list).field_name
```

For example, the following call to function `nth_highest_sal` references the field `salary` in record `emp_info`:

```
DECLARE
   TYPE EmpRec IS RECORD (
      emp_id    NUMBER(4),
      job_title CHAR(14),
      salary    REAL(7,2));
   middle_sal REAL;
   FUNCTION nth_highest_sal (n INTEGER) RETURN EmpRec IS
       emp_info EmpRec;
   BEGIN
      ...
      RETURN emp_info;  -- return record
   END;
BEGIN
   middle_sal := nth_highest_sal(10).salary;  -- call function
```

When calling a parameterless function, use the following syntax:

```
function_name().field_name  -- note empty parameter list
```

To reference nested fields in a record returned by a function, use extended dot notation. The syntax follows:

```
function_name(parameter_list).field_name.nested_field_name
```

For instance, the following call to function `item` references the nested field `minutes` in record `item_info`:

```
DECLARE
   TYPE TimeRec IS RECORD (minutes SMALLINT, hours SMALLINT);
   TYPE AgendaItem IS RECORD (
      priority INTEGER,
      subject  VARCHAR2(100),
      duration TimeRec);
   FUNCTION item (n INTEGER) RETURN AgendaItem IS
      item_info AgendaItem;
   BEGIN
      ...
      RETURN item_info;  -- return record
   END;
BEGIN
   ...
   IF item(3).duration.minutes > 30 THEN ...  -- call function
END;
```

Also, use extended dot notation to reference the attributes of an object stored in a field, as the following example shows:

```
DECLARE
   TYPE FlightRec IS RECORD (
      flight_no    INTEGER,
      plane_id     VARCHAR2(10),
      captain      Employee,  -- declare object
      passengers   PassengerList,  -- declare varray
      depart_time  TimeRec,  -- declare nested record
      airport_code VARCHAR2(10));
   flight FlightRec;
BEGIN
   ...
   IF flight.captain.name = 'H Rawlins' THEN ...
END;
```

Assigning and Comparing Records

You can assign the value of an expression to a specific field in a record using the following syntax:

```
record_name.field_name := expression;
```

In the following example, you convert an employee name to upper case:

```
emp_info.ename := UPPER(emp_info.ename);
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once. This can be done in two ways. First, you can assign one user-defined record to another if they have the same datatype. Having fields that match exactly is not enough. Consider the following example:

```
DECLARE
   TYPE DeptRec IS RECORD (
      dept_num  NUMBER(2),
      dept_name VARCHAR2(14),
      location  VARCHAR2(13));
   TYPE DeptItem IS RECORD (
      dept_num  NUMBER(2),
      dept_name VARCHAR2(14),
   location   VARCHAR2(13));
   dept1_info DeptRec;
   dept2_info DeptItem;
```

```
BEGIN
   ...
   dept1_info := dept2_info;  -- illegal; different datatypes
END;
```

As the next example shows, you can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes:

```
DECLARE
   TYPE DeptRec IS RECORD (
      dept_num  NUMBER(2),
      dept_name CHAR(14),
      location  CHAR(13));
   dept1_info DeptRec;
   dept2_info dept%ROWTYPE;
BEGIN
   SELECT * INTO dept2_info FROM dept WHERE deptno = 10;
   dept1_info := dept2_info;
```

Second, you can use the SELECT or FETCH statement to fetch column values into a record, as the example below shows. The columns in the select-list must appear in the same order as the fields in your record.

```
DECLARE
   TYPE DeptRec IS RECORD (
      dept_num  NUMBER(2),
      dept_name CHAR(14),
      location  CHAR(13));
   dept_info DeptRec;
BEGIN
   SELECT deptno, dname, loc INTO dept_info FROM dept
      WHERE deptno = 20;
 ...
END;
```

However, you cannot use the INSERT statement to insert user-defined records into a database table. So, the following statement is illegal:

```
INSERT INTO dept VALUES (dept_info);  -- illegal
```

Also, you cannot assign a list of values to a record using an assignment statement. Therefore, the following syntax is illegal:

```
record_name := (value1, value2, value3, ...);  -- illegal
```

The example below shows that you can assign one nested record to another if they have the same datatype. Such assignments are allowed even if the enclosing records have different datatypes.

```
DECLARE
   TYPE TimeRec IS RECORD (mins SMALLINT, hrs SMALLINT);
   TYPE MeetingRec IS RECORD (
      day     DATE,
      time_of TimeRec,  -- nested record
      room_no INTEGER(4));
   TYPE PartyRec IS RECORD (
      day     DATE,
      time_of TimeRec,  -- nested record
      place   VARCHAR2(25));
   seminar MeetingRec;
   party   PartyRec;
BEGIN
   ...
   party.time_of := seminar.time_of;
END;
```

Comparing Records

Records cannot be tested for nullity, equality, or inequality. For instance, the following `IF` conditions are illegal:

```
BEGIN
   ...
   IF emp_info IS NULL THEN ...  -- illegal
   IF dept2_info > dept1_info THEN ...  -- illegal
END;
```

Manipulating Records

The datatype `RECORD` lets you collect information about the attributes of something. The information is easy to manipulate because you can refer to the collection as a whole. In the following example, you collect accounting figures from database tables `assets` and `liabilities`, then use ratio analysis to compare the performance of two subsidiary companies:

```
DECLARE
   TYPE FiguresRec IS RECORD (cash REAL, notes REAL, ...);
   sub1_figs FiguresRec;
   sub2_figs FiguresRec;
   FUNCTION acid_test (figs FiguresRec) RETURN REAL IS ...
```

```
BEGIN
   SELECT cash, notes, ... INTO sub1_figs FROM assets, liabilities
      WHERE assets.sub = 1 AND liabilities.sub = 1;
   SELECT cash, notes, ... INTO sub2_figs FROM assets, liabilities
      WHERE assets.sub = 2 AND liabilities.sub = 2;
   IF acid_test(sub1_figs) > acid_test(sub2_figs) THEN ...
   ...
END;
```

Notice how easy it is to pass the collected figures to the function `acid_test`, which computes a financial ratio.

In SQL*Plus, suppose you define object type `Passenger`, as follows:

```
SQL> CREATE TYPE Passenger AS OBJECT(
  2  flight_no NUMBER(3),
  3  name      VARCHAR2(20),
  4  seat      CHAR(5));
```

Next, you define VARRAY type `PassengertList`, which stores `Passenger` objects:

```
SQL> CREATE TYPE PassengerList AS VARRAY(300) OF Passenger;
```

Finally, you create relational table `flights`, which has a column of type `PassengerList`, as follows:

```
SQL> CREATE TABLE flights (
  2  flight_no  NUMBER(3),
  3  gate       CHAR(5),
  4  departure  CHAR(15),
  5  arrival    CHAR(15),
  6  passengers PassengerList);
```

Each item in column `passengers` is a varray that will store the passenger list for a given flight. Now, you can populate database table `flights`, as follows:

```
BEGIN
   INSERT INTO flights
      VALUES(109, '80', 'DFW 6:35PM', 'HOU 7:40PM',
         PassengerList(Passenger(109, 'Paula Trusdale', '13C'),
                       Passenger(109, 'Louis Jemenez', '22F'),
                       Passenger(109, 'Joseph Braun', '11B'), ...));
```

```
   INSERT INTO flights
      VALUES(114, '12B', 'SFO 9:45AM', 'LAX 12:10PM',
         PassengerList(Passenger(114, 'Earl Benton', '23A'),
                       Passenger(114, 'Alma Breckenridge', '10E'),
                       Passenger(114, 'Mary Rizutto', '11C'), ...));
   INSERT INTO flights
      VALUES(27, '34', 'JFK 7:05AM', 'MIA 9:55AM',
         PassengerList(Passenger(27, 'Raymond Kiley', '34D'),
                       Passenger(27, 'Beth Steinberg', '3A'),
                       Passenger(27, 'Jean Lafevre', '19C'), ...));
END;
```

In the example below, you fetch rows from database table `flights` into record `flight_info`. That way, you can treat all the information about a flight, including its passenger list, as a logical unit.

```
DECLARE
   TYPE FlightRec IS RECORD (
      flight_no  NUMBER(3),
      gate       CHAR(5),
      departure  CHAR(15),
      arrival    CHAR(15),
      passengers PassengerList);
   flight_info FlightRec;
   CURSOR c1 IS SELECT * FROM flights;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO flight_info;
      EXIT WHEN c1%NOTFOUND;
      FOR i IN 1..flight_info.passengers.LAST LOOP
         IF flight_info.passengers(i).seat = 'NA' THEN
            DBMS_OUTPUT.PUT_LINE(flight_info.passengers(i).name);
            RAISE seat_not_available;
         END IF;
         ...
      END LOOP;
   END LOOP;
   CLOSE c1;
EXCEPTION
   WHEN seat_not_available THEN
      ...
END;
```

5

Interaction with Oracle

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it. —Samuel Johnson

This chapter helps you harness the power of Oracle. You learn how PL/SQL supports the SQL commands, functions, and operators that let you manipulate Oracle data. You also learn how to manage cursors, use cursor variables, and process transactions.

Major Topics

SQL Support

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 (for more information, see Chapter 10, "Native Dynamic SQL"). In addition, PL/SQL conforms to the current ANSI/ISO SQL standard.

Data Manipulation

To manipulate Oracle data, you use the INSERT, UPDATE, DELETE, SELECT, and LOCK TABLE commands. INSERT adds new rows of data to database tables; UPDATE modifies rows; DELETE removes unwanted rows; SELECT retrieves rows that meet your search criteria; and LOCK TABLE temporarily limits access to a table.

Transaction Control

Oracle is transaction oriented; that is, Oracle uses transactions to ensure data integrity. A *transaction* is a series of SQL data manipulation statements that does a logical unit of work. For example, two UPDATE statements might credit one bank account and debit another.

Simultaneously, Oracle makes permanent or undoes all database changes made by a transaction. If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction. Thus, the database is restored to its former state automatically.

You use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION commands to control transactions. COMMIT makes permanent any database changes made during the current transaction. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction. Used with ROLLBACK, SAVEPOINT undoes part of a transaction. SET TRANSACTION sets transaction properties such as read/write access and isolation level.

SQL Functions

PL/SQL lets you use all the SQL functions including the following aggregate functions, which summarize entire columns of Oracle data: AVG, COUNT, GROUPING, MAX, MIN, STDDEV, SUM, and VARIANCE. Except for COUNT(*), all aggregate functions ignore nulls.

You can use the aggregate functions in SQL statements, but *not* in procedural statements. Aggregate functions operate on entire columns unless you use the SELECT GROUP BY statement to sort returned rows into subgroups. If you omit the GROUP BY clause, the aggregate function treats all returned rows as a single group.

You call an aggregate function using the syntax

```
function_name([ALL | DISTINCT] expression)
```

where expression refers to one or more database columns. If you specify ALL (the default), the aggregate function considers all column values including duplicates. If you specify DISTINCT, the aggregate function considers only distinct values. For example, the following statement returns the number of different job titles in the database table emp:

```
SELECT COUNT(DISTINCT job) INTO job_count FROM emp;
```

The function COUNT lets you specify the asterisk (*) option, which returns the number of rows in a table. For example, the following statement returns the number of rows in table emp:

```
SELECT COUNT(*) INTO emp_count FROM emp;
```

SQL Pseudocolumns

PL/SQL recognizes the following SQL pseudocolumns, which return specific data items: CURRVAL, LEVEL, NEXTVAL, ROWID, and ROWNUM. Pseudocolumns are not actual columns in a table but they behave like columns. For example, you can select values from a pseudocolumn. However, you cannot insert into, update, or delete from a pseudocolumn. Also, pseudocolumns are allowed in SQL statements, but *not* in procedural statements.

CURRVAL and NEXTVAL

A *sequence* is a schema object that generates sequential numbers. When you create a sequence, you can specify its initial value and an increment. CURRVAL returns the current value in a specified sequence.

Before you can reference CURRVAL in a session, you must use NEXTVAL to generate a number. A reference to NEXTVAL stores the current sequence number in CURRVAL. NEXTVAL increments the sequence and returns the next value. To obtain the current or next value in a sequence, you must use dot notation, as follows:

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. However, you can use CURRVAL and NEXTVAL only in a SELECT list, the VALUES clause, and the SET clause. In the following example, you use a sequence to insert the same employee number into two tables:

```
INSERT INTO emp VALUES (empno_seq.NEXTVAL, my_ename, ...);
INSERT INTO sals VALUES (empno_seq.CURRVAL, my_sal, ...);
```

If a transaction generates a sequence number, the sequence is incremented immediately whether you commit or roll back the transaction.

LEVEL

You use LEVEL with the SELECT CONNECT BY statement to organize rows from a database table into a tree structure. LEVEL returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2, grandchildren are level 3, and so on.

You specify the direction in which the query walks the tree (down from the root or up from the branches) with the PRIOR operator. In the START WITH clause, you specify a condition that identifies the root of the tree.

ROWID

ROWID returns the rowid (binary address) of a row in a database table. You can use variables of type UROWID to store rowids in a readable format. In the following example, you declare a variable named row_id for that purpose:

```
DECLARE
   row_id UROWID;
```

When you select or fetch a rowid into a UROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to an 18-byte character string. Then, you can compare the UROWID variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement to identify the latest row fetched from a cursor. For an example, see "Fetching Across Commits" on page 5-49.

ROWNUM

ROWNUM returns a number indicating the order in which a row was selected from a table. The first row selected has a ROWNUM of 1, the second row has a ROWNUM of 2, and so on. If a SELECT statement includes an ORDER BY clause, ROWNUMs are assigned to the retrieved rows *before* the sort is done.

You can use ROWNUM in an UPDATE statement to assign unique values to each row in a table. Also, you can use ROWNUM in the WHERE clause of a SELECT statement to limit the number of rows retrieved, as follows:

```
DECLARE
   CURSOR c1 IS SELECT empno, sal FROM emp
      WHERE sal > 2000 AND ROWNUM < 10;  -- returns 10 rows
```

The value of ROWNUM increases only when a row is retrieved, so the only meaningful use of ROWNUM in a WHERE clause is

```
... 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 *Oracle8i SQL Reference*.

Comparison Operators

Typically, you use comparison operators in the WHERE clause of a data manipulation statement to form *predicates*, which compare one expression to another and always yield TRUE, FALSE, or NULL. You can use all the comparison operators listed below to form predicates. Moreover, you can combine predicates using the logical operators AND, OR, and NOT.

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.

Operator	Description
IS NULL	Tests for nulls.
LIKE	Tests whether a character string matches a specified pattern, which can include wildcards.

Set Operators

Set operators combine the results of two queries into one result. INTERSECT returns all distinct rows selected by both queries. MINUS returns all distinct rows selected by the first query but not by the second. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

Row Operators

Row operators return or reference particular rows. ALL retains duplicate rows in the result of a query or in an aggregate expression. DISTINCT eliminates duplicate rows from the result of a query or from an aggregate expression. PRIOR refers to the parent row of the current row returned by a tree-structured query.

Managing Cursors

PL/SQL uses two types of cursors: *implicit* and *explicit*. PL/SQL declares a cursor implicitly for all SQL data manipulation statements, including queries that return only one row. However, for queries that return more than one row, you must declare an explicit cursor or use a cursor FOR loop.

Explicit Cursors

The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly declare a cursor to process the rows. Moreover, you can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.

You use three commands to control a cursor: OPEN, FETCH, and CLOSE. First, you initialize the cursor with the OPEN statement, which identifies the result set. Then, you use the FETCH statement to retrieve the first row. You can execute FETCH repeatedly until all rows have been retrieved. When the last row has been processed, you release the cursor with the CLOSE statement. You can process several queries in parallel by declaring and opening multiple cursors.

Declaring a Cursor

Forward references are not allowed in PL/SQL. So, you must declare a cursor before referencing it in other statements. When you declare a cursor, you name it and associate it with a specific query using the syntax

```
CURSOR cursor_name [(parameter[, parameter]...)]
   [RETURN return_type] IS select_statement;
```

where `return_type` must represent a record or a row in a database table, and `parameter` stands for the following syntax:

```
cursor_parameter_name [IN] datatype [{:= | DEFAULT} expression]
```

For example, you might declare cursors named `c1` and `c2`, as follows:

```
DECLARE
   CURSOR c1 IS SELECT empno, ename, job, sal FROM emp
      WHERE sal > 2000;
   CURSOR c2 RETURN dept%ROWTYPE IS
      SELECT * FROM dept WHERE deptno = 10;
```

The cursor name is an undeclared identifier, not the name of a PL/SQL variable. You cannot assign values to a cursor name or use it in an expression. However, cursors and variables follow the same scoping rules. Naming cursors after database tables is allowed 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. Therefore, they cannot return values to actual parameters. Also, you cannot impose the constraint `NOT NULL` on a cursor parameter.

As the example below shows, you can initialize cursor parameters to default values. That way, you can pass different numbers of actual parameters to a cursor, accepting or overriding the default values as you please. Also, you can add new formal parameters without having to change every reference to the cursor.

```
DECLARE
   CURSOR c1 (low  INTEGER DEFAULT 0,
              high INTEGER DEFAULT 99) IS SELECT ...
```

The scope of cursor parameters is local to the cursor, meaning that they can be referenced only within the query specified in the cursor declaration. The values of cursor parameters are used by the associated query when the cursor is opened.

Opening a Cursor

Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria. For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. An example of the OPEN statement follows:

```
DECLARE
   CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;
   ...
BEGIN
   OPEN c1;
   ...
END;
```

Rows in the result set are not retrieved when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.

Passing Cursor Parameters

You use the OPEN statement to pass parameters to a cursor. Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. For example, given the cursor declaration

```
DECLARE
   emp_name emp.ename%TYPE;
   salary   emp.sal%TYPE;
   CURSOR c1 (name VARCHAR2, salary NUMBER) IS SELECT ...
```

any of the following statements opens the cursor:

```
OPEN c1(emp_name, 3000);
OPEN c1('ATTLEY', 1500);
OPEN c1(emp_name, salary);
```

In the last example, when the identifier `salary` is used in the cursor declaration, it refers to the formal parameter. But, when it is used in the OPEN statement, it refers to the PL/SQL variable. To avoid confusion, use unique identifiers.

Formal parameters declared with a default value need not have a corresponding actual parameter. They can simply assume their default values when the OPEN statement is executed.

You can associate the actual parameters in an OPEN statement with the formal parameters in a cursor declaration using positional or named notation. (See "Positional and Named Notation" on page 7-13.) The datatypes of each actual parameter and its corresponding formal parameter must be compatible.

Fetching with a Cursor

The FETCH statement retrieves the rows in the result set one at a time. After each fetch, the cursor advances to the next row in the result set. An example follows:

```
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list. Typically, you use the FETCH statement in the following way:

```
LOOP
   FETCH c1 INTO my_record;
   EXIT WHEN c1%NOTFOUND;
   -- process data record
END LOOP;
```

The query can reference PL/SQL variables within its scope. However, any variables in the query are evaluated only when the cursor is opened. In the following example, each retrieved salary is multiplied by 2, even though factor is incremented after every fetch:

```
DECLARE
   my_sal emp.sal%TYPE;
   my_job emp.job%TYPE;
   factor INTEGER := 2;
   CURSOR c1 IS SELECT factor*sal FROM emp WHERE job = my_job;
BEGIN
   ...
   OPEN c1;  -- here factor equals 2
   LOOP
      FETCH c1 INTO my_sal;
      EXIT WHEN c1%NOTFOUND;
      factor := factor + 1;  -- does not affect FETCH
   END LOOP;
END;
```

To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values.

However, you can use a different INTO list on separate fetches with the same cursor. Each fetch retrieves another row and assigns values to the target variables, as the following example shows:

```
DECLARE
   CURSOR c1 IS SELECT ename FROM emp;
   name1 emp.ename%TYPE;
   name2 emp.ename%TYPE;
   name3 emp.ename%TYPE;
BEGIN
   OPEN c1;
   FETCH c1 INTO name1;  -- this fetches first row
   FETCH c1 INTO name2;  -- this fetches second row
   FETCH c1 INTO name3;  -- this fetches third row
   ...
   CLOSE c1;
END;
```

If you fetch past the last row in the result set, the values of the target variables are indeterminate.

Note: Eventually, the FETCH statement must fail to return a row, so when that happens, no exception is raised. To detect the failure, you must use the cursor attribute %FOUND or %NOTFOUND. For more information, see "Using Cursor Attributes" on page 5-34

Closing a Cursor

The CLOSE statement disables the cursor, and the result set becomes undefined. An example of the CLOSE statement follows:

```
CLOSE c1;
```

Once a cursor is closed, you can reopen it. 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. When evaluated, the subquery provides a value or set of values to the statement. Often, subqueries are used in the WHERE clause. For example, the following query returns employees not located in Chicago:

```
DECLARE
   CURSOR c1 IS SELECT empno, ename FROM emp
      WHERE deptno IN (SELECT deptno FROM dept
         WHERE loc <> 'CHICAGO');
```

Using a subquery in the FROM clause, the following query returns the number and name of each department with five or more employees:

```
DECLARE
   CURSOR c1 IS SELECT t1.deptno, dname, "STAFF"
      FROM dept t1, (SELECT deptno, COUNT(*) "STAFF"
         FROM emp GROUP BY deptno) t2
      WHERE t1.deptno = t2.deptno AND "STAFF" >= 5;
```

Whereas a subquery is evaluated only once per table, a *correlated subquery* is evaluated once per row. Consider the query below, which returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the emp table, the correlated subquery computes the average salary for that row's department. The row is returned if that row's salary exceeds the average.

```
DECLARE
   CURSOR c1 IS SELECT deptno, ename, sal FROM emp t
      WHERE sal > (SELECT AVG(sal) FROM emp WHERE t.deptno = deptno)
      ORDER BY deptno;
```

Implicit Cursors

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the SQL cursor.

You cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor. But, you can use cursor attributes to get information about the most recently executed SQL statement. See "Using Cursor Attributes" on page 5-34.

Packaging Cursors

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 the 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_actions AS
   /* Declare cursor spec. */
   CURSOR c1 RETURN emp%ROWTYPE;
   ...
END emp_actions;

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

The cursor spec has no SELECT statement because the RETURN clause defines the datatype of the result value. However, the cursor body must have a SELECT statement and the same RETURN clause as the cursor spec. Also, the number and datatypes of items in the SELECT list and the RETURN clause must match.

Packaged cursors increase flexibility. For instance, you can change the cursor body in the last example, as follows, without having to change the cursor spec:

```
CREATE PACKAGE BODY emp_actions AS
   /* Define cursor body. */
   CURSOR c1 RETURN emp%ROWTYPE IS
      SELECT * FROM emp
         WHERE deptno = 20;  -- new WHERE clause
   ...
END emp_actions;
```

Using Cursor FOR Loops

In most situations that require an explicit cursor, you can simplify coding by using a cursor FOR loop instead of the OPEN, FETCH, and CLOSE statements. A cursor FOR loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor, repeatedly fetches rows of values from the result set into fields in the record, and closes the cursor when all rows have been processed.

Consider the PL/SQL block below, which computes results from an experiment, then stores the results in a temporary table. The FOR loop index c1_rec is implicitly declared as a record. Its fields store all the column values fetched from the cursor c1. Dot notation is used to reference individual fields.

```
-- available online in file 'examp7'
DECLARE
   result temp.col1%TYPE;
   CURSOR c1 IS
      SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
   FOR c1_rec IN c1 LOOP
      /* calculate and store the results */
      result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
      INSERT INTO temp VALUES (result, NULL, NULL);
   END LOOP;
   COMMIT;
END;
```

When the cursor FOR loop is entered, the cursor name cannot belong to a cursor that was already opened by an OPEN statement or by an enclosing cursor FOR loop. Before each iteration of the FOR loop, PL/SQL fetches into the implicitly declared record, which is equivalent to a record declared explicitly as follows:

```
c1_rec c1%ROWTYPE;
```

The record is defined only inside the loop. You cannot refer to its fields outside the loop. For example, the following reference is illegal:

```
FOR c1_rec IN c1 LOOP
   ...
END LOOP;
result := c1_rec.n2 + 3;  -- illegal
```

The sequence of statements inside the loop is executed once for each row that satisfies the query associated with the cursor. When you leave the loop, the cursor is closed automatically—even if you use an EXIT or GOTO statement to leave the loop prematurely or an exception is raised inside the loop.

Using Subqueries

You need not declare a cursor because PL/SQL lets you substitute a subquery. The following cursor FOR loop calculates a bonus, then inserts the result into a database table:

```
DECLARE
   bonus REAL;
BEGIN
   FOR emp_rec IN (SELECT empno, sal, comm FROM emp) LOOP
      bonus := (emp_rec.sal * 0.05) + (emp_rec.comm * 0.25);
      INSERT INTO bonuses VALUES (emp_rec.empno, bonus);
   END LOOP;
   COMMIT;
END;
```

Using Aliases

Fields in the implicitly declared record hold column values from the most recently fetched row. The fields have the same names as corresponding columns in the SELECT list. But, what happens if a select item is an expression? Consider the following example:

```
CURSOR c1 IS
   SELECT empno, sal+NVL(comm,0), job FROM ...
```

In such cases, you must include an alias for the select item. In the following example, wages is an alias for the select item sal+NVL(comm,0):

```
CURSOR c1 IS
   SELECT empno, sal+NVL(comm,0) wages, job FROM ...
```

To reference the corresponding field, use the alias instead of a column name, as follows:

```
IF emp_rec.wages < 1000 THEN ...
```

Passing Parameters

You can pass parameters to the cursor in a cursor FOR loop. In the following example, you pass a department number. Then, you compute the total wages paid to employees in that department. Also, you determine how many employees have salaries higher than $2000 and / or commissions larger than their salaries.

```
-- available online in file 'examp8'
DECLARE
   CURSOR emp_cursor(dnum NUMBER) IS
      SELECT sal, comm FROM emp WHERE deptno = dnum;
   total_wages NUMBER(11,2) := 0;
   high_paid   NUMBER(4) := 0;
   higher_comm NUMBER(4) := 0;
BEGIN
   /* The number of iterations will equal the number of rows
      returned by emp_cursor. */
   FOR emp_record IN emp_cursor(20) LOOP
      emp_record.comm := NVL(emp_record.comm, 0);
      total_wages := total_wages + emp_record.sal +
         emp_record.comm;
      IF emp_record.sal > 2000.00 THEN
         high_paid := high_paid + 1;
      END IF;
      IF emp_record.comm > emp_record.sal THEN
         higher_comm := higher_comm + 1;
      END IF;
   END LOOP;
   INSERT INTO temp VALUES (high_paid, higher_comm,
      'Total Wages: ' || TO_CHAR(total_wages));
   COMMIT;
END;
```

Using Cursor Variables

Like a cursor, a cursor variable points to the current row in the result set of a multi-row query. But, cursors differ from cursor variables the way constants differ from variables. Whereas a cursor is static, a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query. This gives you more flexibility.

Also, you can assign new values to a cursor variable and pass it as a parameter to local and stored subprograms. This gives you an easy way to centralize data retrieval.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as input host variable (bind variable) to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side.

The Oracle server also has a PL/SQL engine. So, you can pass cursor variables back and forth between an application and server via remote procedure calls (RPCs).

What Are Cursor Variables?

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, *not* an item. In PL/SQL, a pointer has datatype REF X, where REF is short for REFERENCE and X stands for a class of objects. Therefore, a cursor variable has datatype REF CURSOR.

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use an explicit cursor, which names the work area. Or, you can use a cursor variable, which points to the work area. Whereas a cursor always refers to the same query work area, a cursor variable can refer to different work areas. So, cursors and cursor variables are *not* interoperable; that is, you cannot use one where the other is expected.

Why Use Cursor Variables?

Mainly, you use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single round trip.

Defining REF CURSOR Types

To create cursor variables, you take two steps. First, you define a REF CURSOR type, then declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package using the syntax

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;
```

where `ref_type_name` is a type specifier used in subsequent declarations of cursor variables and `return_type` must represent a record or a row in a database table. In the following example, you specify a return type that represents a row in the database table `dept`:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
```

REF CURSOR types can be *strong* (restrictive) or *weak* (nonrestrictive). As the next example shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;  -- strong
   TYPE GenericCurTyp IS REF CURSOR;  -- weak
```

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Declaring Cursor Variables

Once you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable `dept_cv`:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

Note: You cannot declare cursor variables in a package. Unlike packaged variables, cursor variables do not have persistent state. Remember, declaring a cursor variable creates a pointer, not an item. So, cursor variables cannot be saved in the database.

Cursor variables follow the usual scoping and instantiation rules. Local PL/SQL cursor variables are instantiated when you enter a block or subprogram and cease to exist when you exit.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable, as follows:

```
DECLARE
   TYPE TmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   tmp_cv TmpCurTyp;  -- declare cursor variable
   TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Likewise, you can use %TYPE to provide the datatype of a record variable, as the following example shows:

```
DECLARE
   dept_rec dept%ROWTYPE;  -- declare record variable
   TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      empno NUMBER(4),
      ename VARCHAR2(10),
      sal   NUMBER(7,2));
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following example, you define the REF CURSOR type EmpCurTyp, then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

Caution: Like all pointers, cursor variables increase the possibility of parameter aliasing. For more information, see "Parameter Aliasing" on page 7-21.

Controlling Cursor Variables

You use three statements to control a cursor variable: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set one at a time. When all the rows are processed, you CLOSE the cursor variable.

Opening a Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi-row query, executes the query, and identifies the result set. Here is the syntax:

```
OPEN {cursor_variable_name | :host_cursor_variable_name}
   FOR select_statement;
```

where host_cursor_variable_name identifies a cursor variable declared in a PL/SQL host environment such as an OCI or Pro*C program.

Unlike cursors, cursor variables take no parameters. However, no flexibility is lost because you can pass whole queries (not just parameters) to a cursor variable. The query can reference host variables and PL/SQL variables, parameters, and functions but cannot be FOR UPDATE.

In the example below, you open the cursor variable emp_cv. Notice that you can apply cursor attributes (%FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT) to a cursor variable.

```
IF NOT emp_cv%ISOPEN THEN
   /* Open cursor variable. */
   OPEN emp_cv FOR SELECT * FROM emp;
END IF;
```

Other OPEN-FOR statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. (Recall that consecutive OPENs of a static cursor raise the predefined exception CURSOR_ALREADY_OPEN.) When you reopen a cursor variable for a different query, the previous query is lost.

Typically, you open a cursor variable by passing it to a stored procedure that declares a cursor variable as one of its formal parameters. For example, the following packaged procedure opens the cursor variable emp_cv:

```
CREATE PACKAGE emp_data AS
   ...
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp);
END emp_data;

CREATE PACKAGE BODY emp_data AS
   ...
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
   BEGIN
      OPEN emp_cv FOR SELECT * FROM emp;
   END open_emp_cv;
END emp_data;
```

When you declare a cursor variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the IN OUT mode. That way, the subprogram can pass an open cursor back to the caller.

Alternatively, you can use a stand-alone procedure to open the cursor variable. Simply define the REF CURSOR type in a separate package, then reference that type in the stand-alone procedure. For instance, if you create the following bodiless package, you can create stand-alone procedures that reference the types it defines:

```
CREATE PACKAGE cv_types AS
   TYPE GenericCurTyp IS REF CURSOR;
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
   ...
END cv_types;
```

In the next example, you create a stand-alone procedure that references the REF CURSOR type EmpCurTyp, which is defined in the package cv_types:

```
CREATE PROCEDURE open_emp_cv (emp_cv IN OUT cv_types.EmpCurTyp) AS
BEGIN
   OPEN emp_cv FOR SELECT * FROM emp;
END open_emp_cv;
```

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. (In this context, a *selector* is a variable used to select one of several alternatives in a conditional control statement.) When called, the procedure opens the cursor variable emp_cv for the chosen query.

```
CREATE PACKAGE emp_data AS
   TYPE GenericCurTyp IS REF CURSOR;
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp, choice NUMBER);
END emp_data;

CREATE PACKAGE BODY emp_data AS
   PROCEDURE open_emp_cv (
      emp_cv IN OUT EmpCurTyp,
      choice NUMBER) IS
   BEGIN
      IF choice = 1 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
      ELSIF choice = 2 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
      ELSIF choice = 3 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
      END IF;
   END open_emp_cv;
END emp_data;
```

For more flexibility, you can pass a cursor variable and a selector to a stored procedure that executes queries with different return types. Consider this example:

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

Using 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 one at a time from the result set of a multi-row query. Here is the syntax:

```
FETCH {cursor_variable_name | :host_cursor_variable_name}
   INTO {variable_name[, variable_name]... | record_name};
```

In the next example, you fetch rows from the cursor variable emp_cv into the user-defined record emp_rec:

```
LOOP
   /* Fetch from cursor variable. */
   FETCH emp_cv INTO emp_rec;
   EXIT WHEN emp_cv%NOTFOUND; -- exit when last row is fetched
   -- process data record
END LOOP;
```

Any variables in the associated query are evaluated only when the cursor variable is opened. To change the result set or the values of variables in the query, you must reopen the cursor variable with the variables set to their new values. However, you can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set.

PL/SQL makes sure the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible field or variable in the INTO clause. Also, the number of fields or variables must equal the number of column values. Otherwise, you get an error.

The error occurs at compile time if the cursor variable is strongly typed or at run time if it is weakly typed. At run time, PL/SQL raises the predefined exception ROWTYPE_MISMATCH *before* the first fetch. So, if you trap the error and execute the FETCH statement using a different INTO clause, no rows are lost.

When you declare a cursor variable as the formal parameter of a subprogram that fetches from the cursor variable, you must specify the IN or IN OUT mode. However, if the subprogram also opens the cursor variable, you must specify the IN OUT mode.

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Closing a Cursor Variable

The CLOSE statement disables a cursor variable. After that, the associated result set is undefined. Here is the syntax:

```
CLOSE {cursor_variable_name | :host_cursor_variable_name);
```

In the following example, when the last row is processed, you close the cursor variable emp_cv:

```
LOOP
   FETCH emp_cv INTO emp_rec;
   EXIT WHEN emp_cv%NOTFOUND;
   -- process data record
END LOOP;
/* Close cursor variable. */
CLOSE emp_cv;
```

When declaring a cursor variable as the formal parameter of a subprogram that closes the cursor variable, you must specify the IN or IN OUT mode.

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Example 1

Consider the stored procedure below, which searches the database of a main library for books, periodicals, and tapes. A master table stores the title and category code (where 1 = book, 2 = periodical, 3 = tape) of each item. Three detail tables store category-specific information. When called, the procedure searches the master table by title, uses the associated category code to pick an OPEN-FOR statement, then opens a cursor variable for a query of the proper detail table.

```
CREATE PACKAGE cv_types AS
   TYPE LibCurTyp IS REF CURSOR;
   ...
END cv_types;

CREATE PROCEDURE find_item (title  VARCHAR2(100),
                           lib_cv IN OUT cv_types.LibCurTyp) AS
   code BINARY_INTEGER;
BEGIN
   SELECT item_code FROM titles INTO code
      WHERE item_title = title;
```

```
   IF code = 1 THEN
      OPEN lib_cv FOR SELECT * FROM books
         WHERE book_title = title;
   ELSIF code = 2 THEN
      OPEN lib_cv FOR SELECT * FROM periodicals
         WHERE periodical_title = title;
   ELSIF code = 3 THEN
      OPEN lib_cv FOR SELECT * FROM tapes
         WHERE tape_title = title;
   END IF;
END find_item;
```

Example 2

A client-side application in a branch library might use the following PL/SQL block to display the retrieved information:

```
DECLARE
   lib_cv         cv_types.LibCurTyp;
   book_rec       books%ROWTYPE;
   periodical_rec periodicals%ROWTYPE;
   tape_rec       tapes%ROWTYPE;
BEGIN
   get_title(:title);  -- title is a host variable
   find_item(:title, lib_cv);
   FETCH lib_cv INTO book_rec;
   display_book(book_rec);
EXCEPTION
   WHEN ROWTYPE_MISMATCH THEN
      BEGIN
         FETCH lib_cv INTO periodical_rec;
         display_periodical(periodical_rec);
      EXCEPTION
         WHEN ROWTYPE_MISMATCH THEN
            FETCH lib_cv INTO tape_rec;
            display_tape(tape_rec);
      END;
END;
```

Example 3

The following Pro*C program prompts the user to select a database table, opens a cursor variable for a query of that table, then fetches rows returned by the query:

```
#include <stdio.h>
#include <sqlca.h>
void sql_error();
main()
{
   char temp[32];
   EXEC SQL BEGIN DECLARE SECTION;
      char * uid = "scott/tiger";
      SQL_CURSOR generic_cv;   /* cursor variable */
      int    table_num;        /* selector        */
      struct                   /* EMP record      */
      {
         int   emp_num;
         char  emp_name[11];
         char  job_title[10];
         int   manager;
         char  hire_date[10];
         float salary;
         float commission;
         int   dept_num;
      } emp_rec;
      struct                   /* DEPT record     */
      {
         int   dept_num;
         char  dept_name[15];
         char  location[14];
      } dept_rec;
      struct                   /* BONUS record    */
      {
         char  emp_name[11];
         char  job_title[10];
         float salary;
      } bonus_rec;
   EXEC SQL END DECLARE SECTION;
   /* Handle Oracle errors. */
   EXEC SQL WHENEVER SQLERROR DO sql_error();

   /* Connect to Oracle. */
   EXEC SQL CONNECT :uid;
```

```
/* Initialize cursor variable. */
EXEC SQL ALLOCATE :generic_cv;

/* Exit loop when done fetching. */
EXEC SQL WHENEVER NOT FOUND DO break;

for (;;)
{
   printf("\n1 = EMP, 2 = DEPT, 3 = BONUS");
   printf("\nEnter table number (0 to quit): ");
   gets(temp);
   table_num = atoi(temp);
   if (table_num <= 0) break;

   /* Open cursor variable. */
   EXEC SQL EXECUTE
      BEGIN
         IF :table_num = 1 THEN
            OPEN :generic_cv FOR SELECT * FROM emp;
         ELSIF :table_num = 2 THEN
            OPEN :generic_cv FOR SELECT * FROM dept;
         ELSIF :table_num = 3 THEN
            OPEN :generic_cv FOR SELECT * FROM bonus;
         END IF;
      END;
   END-EXEC;
   for (;;)
   {
      switch (table_num)
      {
         case 1:  /* Fetch row into EMP record. */
            EXEC SQL FETCH :generic_cv INTO :emp_rec;
            break;
         case 2:  /* Fetch row into DEPT record. */
            EXEC SQL FETCH :generic_cv INTO :dept_rec;
            break;
         case 3:  /* Fetch row into BONUS record. */
            EXEC SQL FETCH :generic_cv INTO :bonus_rec;
            break;
      }
      /* Process data record here. */
   }
   /* Close cursor variable. */
   EXEC SQL CLOSE :generic_cv;
}
```

```
   exit(0);
}
void sql_error()
{
   /* Handle SQL error here. */
}
```

Example 4

A host variable is a variable you declare in a host environment, then pass to one or more PL/SQL programs, which can use it like any other variable. In the SQL*Plus environment, to declare a host variable, use the command VARIABLE. For example, you declare a variable of type NUMBER as follows:

```
VARIABLE return_code NUMBER
```

Both SQL*Plus and PL/SQL can reference the host variable, and SQL*Plus can display its value.

Note: If you declare a host variable with the same name as a PL/SQL program variable, the latter takes precedence.

To reference a host variable in PL/SQL, you must prefix its name with a colon (:), as the following example shows:

```
BEGIN
   :return_code := 0;
   IF credit_check_ok(acct_no) THEN
      :return_code := 1;
   END IF;
   ...
END;
```

To display the value of a host variable in SQL*Plus, use the PRINT command, as follows:

```
SQL> PRINT return_code

RETURN_CODE
-----------
          1
```

The SQL*Plus datatype REFCURSOR lets you declare cursor variables, which you can use to return query results from stored subprograms. In the script below, you declare a host variable of type REFCURSOR. You use the SQL*Plus command SET AUTOPRINT ON to display the query results automatically.

```
CREATE PACKAGE emp_data AS
   TYPE EmpRecTyp IS RECORD (
      emp_id    NUMBER(4),
      emp_name  CHAR(10),
      job_title CHAR(9),
      dept_name CHAR(14),
      dept_loc  CHAR(13));
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   PROCEDURE get_staff (
      dept_no IN NUMBER,
      emp_cv IN OUT EmpCurTyp);
END;
/
CREATE PACKAGE BODY emp_data AS
   PROCEDURE get_staff (
      dept_no IN NUMBER,
      emp_cv IN OUT EmpCurTyp) IS
   BEGIN
      OPEN emp_cv FOR
         SELECT empno, ename, job, dname, loc FROM emp, dept
            WHERE emp.deptno = dept_no AND emp.deptno = dept.deptno
            ORDER BY empno;
   END;
END;
/
COLUMN EMPNO HEADING Number
COLUMN ENAME HEADING Name
COLUMN JOB HEADING JobTitle
COLUMN DNAME HEADING Department
COLUMN LOC HEADING Location
SET AUTOPRINT ON

VARIABLE cv REFCURSOR
EXECUTE emp_data.get_staff(20, :cv)
```

Reducing Network Traffic

When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping OPEN-FOR statements. For example, the following PL/SQL block opens five cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
   OPEN :emp_cv FOR SELECT * FROM emp;
   OPEN :dept_cv FOR SELECT * FROM dept;
   OPEN :grade_cv FOR SELECT * FROM salgrade;
   OPEN :pay_cv FOR SELECT * FROM payroll;
   OPEN :ins_cv FOR SELECT * FROM insurance;
END;
```

This might be useful in Oracle Forms, for instance, when you want to populate a multi-block form.

When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes. That allows your OCI or Pro*C program to use these work areas for ordinary cursor operations. In the following example, you open several such work areas in a single round trip:

```
BEGIN
   OPEN :c1 FOR SELECT 1 FROM dual;
   OPEN :c2 FOR SELECT 1 FROM dual;
   OPEN :c3 FOR SELECT 1 FROM dual;
   OPEN :c4 FOR SELECT 1 FROM dual;
   OPEN :c5 FOR SELECT 1 FROM dual;
   ...
END;
```

The cursors assigned to c1, c2, c3, c4, and c5 behave normally, and you can use them for any purpose. When finished, simply release the cursors, as follows:

```
BEGIN
   CLOSE :c1;
   CLOSE :c2;
   CLOSE :c3;
   CLOSE :c4;
   CLOSE :c5;
   ...
END;
```

Avoiding Errors

If both cursor variables involved in an assignment are strongly typed, they must have the same datatype. In the following example, even though the cursor variables have the same return type, the assignment raises an exception because they have different datatypes:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   TYPE TmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (
      emp_cv IN OUT EmpCurTyp,
      tmp_cv IN OUT TmpCurTyp) IS
   BEGIN
      ...
      emp_cv := tmp_cv;  -- causes 'wrong type' error
   END;
```

However, if one or both cursor variables are weakly typed, they need not have the same datatype.

If you try to fetch from, close, or apply cursor attributes to a cursor variable that does not point to a query work area, PL/SQL raises INVALID_CURSOR. You can make a cursor variable (or parameter) point to a query work area in two ways:

- OPEN the cursor variable FOR the query.
- Assign to the cursor variable the value of an already OPENed host cursor variable or PL/SQL cursor variable.

The following example shows how these ways interact:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   emp_cv1 EmpCurTyp;
   emp_cv2 EmpCurTyp;
   emp_rec emp%ROWTYPE;
BEGIN
   /* The following assignment is useless because emp_cv1
      does not point to a query work area yet. */
   emp_cv2 := emp_cv1;  -- useless
   /* Make emp_cv1 point to a query work area. */
   OPEN emp_cv1 FOR SELECT * FROM emp;
   /* Use emp_cv1 to fetch first row from emp table. */
   FETCH emp_cv1 INTO emp_rec;
   /* The following fetch raises an exception because emp_cv2
      does not point to a query work area yet. */
```

```
   FETCH emp_cv2 INTO emp_rec;  -- raises INVALID_CURSOR
EXCEPTION
   WHEN INVALID_CURSOR THEN
      /* Make emp_cv1 and emp_cv2 point to same work area. */
      emp_cv2 := emp_cv1;
      /* Use emp_cv2 to fetch second row from emp table. */
      FETCH emp_cv2 INTO emp_rec;
      /* Reuse work area for another query. */
      OPEN emp_cv2 FOR SELECT * FROM old_emp;
      /* Use emp_cv1 to fetch first row from old_emp table.
         The following fetch succeeds because emp_cv1 and
         emp_cv2 point to the same query work area. */
      FETCH emp_cv1 INTO emp_rec;  -- succeeds
END;
```

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises ROWTYPE_MISMATCH if the return types of the actual and formal parameters are incompatible.

In the Pro*C example below, you define a packaged REF CURSOR type, specifying the return type emp%ROWTYPE. Next, you create a stand-alone procedure that references the new type. Then, inside a PL/SQL block, you open a host cursor variable for a query of the dept table. Later, when you pass the open host cursor variable to the stored procedure, PL/SQL raises ROWTYPE_MISMATCH because the return types of the actual and formal parameters are incompatible.

```
CREATE PACKAGE cv_types AS
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   ...
END cv_types;
/
CREATE PROCEDURE open_emp_cv (emp_cv IN OUT cv_types.EmpCurTyp) AS
BEGIN
   OPEN emp_cv FOR SELECT * FROM emp;
END open_emp_cv;
/
-- anonymous PL/SQL block in Pro*C program
EXEC SQL EXECUTE
   BEGIN
      OPEN :cv FOR SELECT * FROM dept;
      ...
      open_emp_cv(:cv); -- raises ROWTYPE_MISMATCH
   END;
END-EXEC;
```

Restrictions on Cursor Variables

Currently, cursor variables are subject to the following restrictions:

- You cannot declare cursor variables in a package. For example, the following declaration is illegal:

```
CREATE PACKAGE emp_stuff AS
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   emp_cv IN OUT EmpCurTyp;  -- illegal
END emp_stuff;
```

- Remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use RPCs to pass cursor variables from one server to another.
- 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.
- The query associated with a cursor variable in an OPEN-FOR statement cannot be FOR UPDATE.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
- You cannot assign nulls to a cursor variable.
- You cannot use REF CURSOR types to specify column types in a CREATE TABLE or CREATE VIEW statement. So, database columns cannot store the values of cursor variables.
- You cannot use a REF CURSOR type to specify the element type of a collection, which means that elements in a index-by table, nested table, or varray cannot store the values of cursor variables.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected. For example, the following cursor FOR loop is illegal:

```
DECLARE
   CURSOR emp_cur IS SELECT * FROM emp;  -- static cursor
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   emp_cv  EmpCurTyp;  -- cursor variable
BEGIN
   ...
   FOR emp_rec IN emp_cv LOOP ...  -- illegal
END;
```

Using Cursor Attributes

Each cursor or 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.

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 one at a time.

%FOUND

After a cursor or cursor variable is opened but before the first fetch, %FOUND yields NULL. Thereafter, it yields TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row. In the following example, you use %FOUND to select an action:

```
LOOP
   FETCH c1 INTO my_ename, my_sal, my_hiredate;
   IF c1%FOUND THEN  -- fetch succeeded
      ...
   ELSE  -- fetch failed, so exit loop
      EXIT;
   END IF;
END LOOP;
```

If a cursor or cursor variable is not open, referencing it with %FOUND raises the predefined exception INVALID_CURSOR.

%ISOPEN

%ISOPEN yields TRUE if its cursor or cursor variable is open; otherwise, %ISOPEN yields FALSE. In the following example, you use %ISOPEN to select an action:

```
IF c1%ISOPEN THEN  -- cursor is open
   ...
ELSE  -- cursor is closed, so open it
   OPEN c1;
END IF;
```

%NOTFOUND

%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields FALSE if the last fetch returned a row, or TRUE if the last fetch failed to return a row. In the following example, you use %NOTFOUND to exit a loop when FETCH fails to return a row:

```
LOOP
   FETCH c1 INTO my_ename, my_sal, my_hiredate;
   EXIT WHEN c1%NOTFOUND;
   ...
END LOOP;
```

Before the first fetch, %NOTFOUND evaluates to NULL. So, if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement instead:

```
EXIT WHEN c1%NOTFOUND OR ci%NOTFOUND IS NULL;
```

If a cursor or cursor variable is not open, referencing it with %NOTFOUND raises INVALID_CURSOR.

%ROWCOUNT

When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. In the next example, you use %ROWCOUNT to take action if more than ten rows have been fetched:

```
LOOP
   FETCH c1 INTO my_ename, my_deptno;
   IF c1%ROWCOUNT > 10 THEN
      ...
   END IF;
   ...
END LOOP;
```

If a cursor or cursor variable is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR.

Table 5–1 shows what each cursor attribute yields before and after you execute an OPEN, FETCH, or CLOSE statement.

Table 5–1 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
	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.

Some Examples

Suppose you have a table named data_table that holds data collected from laboratory experiments, and you want to analyze the data from experiment 1. In the following example, you compute the results and store them in a database table named temp:

```
-- available online in file 'examp5'
DECLARE
   num1   data_table.n1%TYPE;  -- Declare variables
   num2   data_table.n2%TYPE;  -- having same types as
   num3   data_table.n3%TYPE;  -- database columns
   result temp.col1%TYPE;
   CURSOR c1 IS
      SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
```

```
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO num1, num2, num3;
      EXIT WHEN c1%NOTFOUND;  -- TRUE when FETCH finds no more rows
      result := num2/(num1 + num3);
      INSERT INTO temp VALUES (result, NULL, NULL);
   END LOOP;
   CLOSE c1;
   COMMIT;
END;
```

In the next example, you check all storage bins that contain part number 5469, withdrawing their contents until you accumulate 1000 units:

```
-- available online in file 'examp6'
DECLARE
   CURSOR bin_cur(part_number NUMBER) IS
      SELECT amt_in_bin FROM bins
         WHERE part_num = part_number AND amt_in_bin > 0
         ORDER BY bin_num
         FOR UPDATE OF amt_in_bin;
   bin_amt        bins.amt_in_bin%TYPE;
   total_so_far   NUMBER(5) := 0;
   amount_needed  CONSTANT NUMBER(5) := 1000;
   bins_looked_at NUMBER(3) := 0;
BEGIN
   OPEN bin_cur(5469);
   WHILE total_so_far < amount_needed LOOP
      FETCH bin_cur INTO bin_amt;
      EXIT WHEN bin_cur%NOTFOUND;
         -- if we exit, there's not enough to fill the order
      bins_looked_at := bins_looked_at + 1;
      IF total_so_far + bin_amt < amount_needed THEN
         UPDATE bins SET amt_in_bin = 0
            WHERE CURRENT OF bin_cur;
               -- take everything in the bin
         total_so_far := total_so_far + bin_amt;
      ELSE  -- we finally have enough
         UPDATE bins SET amt_in_bin = amt_in_bin
            - (amount_needed - total_so_far)
            WHERE CURRENT OF bin_cur;
         total_so_far := amount_needed;
      END IF;
   END LOOP;
```

```
   CLOSE bin_cur;
   INSERT INTO temp
      VALUES (NULL, bins_looked_at, '<- bins looked at');
   COMMIT;
END;
```

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.

%FOUND

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

Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE.

%NOTFOUND

%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

%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

The values of the cursor attributes always refer to the most recently executed SQL statement, wherever that statement is. It might be in a different scope (for example, in a sub-block). So, if you want to save an attribute value for later use, assign it to a Boolean variable immediately. In the following example, relying on the IF condition is dangerous because the procedure check_status might have changed the value of %NOTFOUND:

```
BEGIN
   ...
   UPDATE parts SET quantity = quantity - 1 WHERE partno = part_id;
   check_status(part_id);  -- procedure call
   IF SQL%NOTFOUND THEN  -- dangerous!
      ...
   END;
END;
```

You can improve the code as follows:

```
BEGIN
   ...
   UPDATE parts SET quantity = quantity - 1 WHERE partno = part_id;
   sql_notfound := SQL%NOTFOUND; -- assign value to Boolean variable
   check_status(part_id);
   IF sql_notfound THEN ...
END;
```

If a SELECT INTO statement fails to return a row, PL/SQL raises the predefined exception NO_DATA_FOUND whether you check %NOTFOUND on the next line or not. Consider the following example:

```
BEGIN
   ...
   SELECT sal INTO my_sal FROM emp WHERE empno = my_empno;
      -- might raise NO_DATA_FOUND
   IF SQL%NOTFOUND THEN  -- condition tested only when false
      ...  -- this action is never taken
   END IF;
```

The check is useless because the IF condition is tested only when %NOTFOUND is false. When PL/SQL raises NO_DATA_FOUND, normal execution stops and control transfers to the exception-handling part of the block.

However, a SELECT INTO statement that calls a SQL aggregate function never raises NO_DATA_FOUND because aggregate functions always return a value or a null. In such cases, %NOTFOUND yields FALSE, as the following example shows:

```
BEGIN
   ...
   SELECT MAX(sal) INTO my_sal FROM emp WHERE deptno = my_deptno;
      -- never raises NO_DATA_FOUND
   IF SQL%NOTFOUND THEN  -- always tested but never true
      ...  -- this action is never taken
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN ...  -- never invoked
```

Processing Transactions

This section explains how to do transaction processing. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to Oracle data are made permanent or undone.

The jobs or tasks that Oracle manages are called *sessions*. A *user session* is started when you run an application program or an Oracle tool and connect to Oracle. To allow user sessions to work "simultaneously" and share computer resources, Oracle must control *concurrency*, the accessing of the same data by many users. Without adequate concurrency controls, there might be a loss of *data integrity*. That is, changes to data might be made in the wrong order.

Oracle uses *locks* to control concurrent access to data. A lock gives you temporary ownership of a database resource such as a table or row of data. Thus, data cannot be changed by other users until you finish with it. You need never explicitly lock a resource because default locking mechanisms protect Oracle data and structures. However, you can request *data locks* on tables or rows when it is to your advantage to override default locking. You can choose from several *modes* of locking such as *row share* and *exclusive*.

A *deadlock* can occur when two or more users try to access the same schema object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until Oracle breaks the deadlock by signaling an error to the last participating transaction.

When a table is being queried by one user and updated by another at the same time, Oracle generates a *read-consistent* view of the data for the query. That is, once a query begins and as it proceeds, the data read by the query does not change. As update activity continues, Oracle takes *snapshots* of the table's data and records changes in a *rollback segment*. Oracle uses rollback segments to build read-consistent query results and to undo changes if necessary.

How Transactions Guard Your Database

A transaction is a series of SQL data manipulation statements that does a logical unit of work. Oracle treats the series of SQL statements as a unit so that all the changes brought about by the statements are either *committed* (made permanent) or *rolled back* (undone) at the same time. If your program fails in the middle of a transaction, the database is automatically restored to its former state.

The first SQL statement in your program begins a transaction. When one transaction ends, the next SQL statement automatically begins another transaction. Thus, every SQL statement is part of a transaction. A *distributed transaction* includes at least one SQL statement that updates data at multiple nodes in a distributed database.

The `COMMIT` and `ROLLBACK` statements ensure that all database changes brought about by SQL operations are either made permanent or undone at the same time. All the SQL statements executed since the last commit or rollback make up the current transaction. The `SAVEPOINT` statement names and marks the current point in the processing of a transaction.

Using COMMIT

The COMMIT statement ends the current transaction and makes permanent any changes made during that transaction. Until you commit the changes, other users cannot access the changed data; they see the data as it was before you made the changes.

Consider a simple transaction that transfers money from one bank account to another. The transaction requires two updates because it debits the first account, then credits the second. In the example below, after crediting the second account, you issue a commit, which makes the changes permanent. Only then do other users see the changes.

```
BEGIN
   ...
   UPDATE accts SET bal = my_bal - debit
      WHERE acctno = 7715;
   ...
   UPDATE accts SET bal = my_bal + credit
      WHERE acctno = 7720;
   COMMIT WORK;
END;
```

The COMMIT statement releases all row and table locks. It also erases any savepoints (discussed later) marked since the last commit or rollback. The optional keyword WORK has no effect other than to improve readability. The keyword END signals the end of a PL/SQL block, *not* the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.

The COMMENT clause lets you specify a comment to be associated with a distributed transaction. When you issue a commit, changes to each database affected by a distributed transaction are made permanent. However, if a network or machine fails during the commit, the state of the distributed transaction might be unknown or *in doubt*. In that case, Oracle stores the text specified by COMMENT in the data dictionary along with the transaction ID. The text must be a quoted literal up to 50 characters long. An example follows:

```
COMMIT COMMENT 'In-doubt order transaction; notify Order Entry';
```

PL/SQL does not support the FORCE clause, which, in SQL, manually commits an in-doubt distributed transaction. For example, the following COMMIT statement is illegal:

```
COMMIT FORCE '23.51.54';  -- illegal
```

Using ROLLBACK

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. Rolling back is useful for two reasons. First, if you make a mistake like deleting the wrong row from a table, a rollback restores the original data. Second, if you start a transaction that you cannot finish because an exception is raised or a SQL statement fails, a rollback lets you return to the starting point to take corrective action and perhaps try again.

Consider the example below, in which you insert information about an employee into three different database tables. All three tables have a column that holds employee numbers and is constrained by a unique index. If an INSERT statement tries to store a duplicate employee number, the predefined exception DUP_VAL_ON_INDEX is raised. In that case, you want to undo all changes, so you issue a rollback in the exception handler.

```
DECLARE
   emp_id  INTEGER;
   ...
BEGIN
   SELECT empno, ... INTO emp_id, ... FROM new_emp WHERE ...
   ...
   INSERT INTO emp VALUES (emp_id, ...);
   INSERT INTO tax VALUES (emp_id, ...);
   INSERT INTO pay VALUES (emp_id, ...);
   ...
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK;
   ...
END;
```

Statement-Level Rollbacks

Before executing a SQL statement, Oracle marks an implicit savepoint. Then, if the statement fails, Oracle rolls it back automatically. For example, if an INSERT statement raises an exception by trying to insert a duplicate value in a unique index, the statement is rolled back. Only work started by the failed SQL statement is lost. Work done before that statement in the current transaction is kept.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Before executing a SQL statement, Oracle must *parse* it, that is, examine it to make sure it follows syntax rules and refers to valid schema objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

Using SAVEPOINT

SAVEPOINT names and marks the current point in the processing of a transaction. Used with the ROLLBACK TO statement, savepoints let you undo parts of a transaction instead of the whole transaction. In the example below, you mark a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the empno column, the predefined exception DUP_VAL_ON_INDEX is raised. In that case, you roll back to the savepoint, undoing just the insert.

```
DECLARE
   emp_id  emp.empno%TYPE;
BEGIN
   UPDATE emp SET ... WHERE empno = emp_id;
   DELETE FROM emp WHERE ...
   ...
   SAVEPOINT do_insert;
   INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK TO do_insert;
END;
```

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. However, the savepoint to which you roll back is not erased. For example, if you mark five savepoints, then roll back to the third, only the fourth and fifth are erased. A simple rollback or commit erases all savepoints.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent. However, you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers and can be reused within a transaction. This moves the savepoint from its old position to the current point in the transaction. Thus, a rollback to the savepoint affects only the current part of your transaction. An example follows:

```
BEGIN
   ...
   SAVEPOINT my_point;
   UPDATE emp SET ... WHERE empno = emp_id;
```

```
   ...
   SAVEPOINT my_point;  -- move my_point to current point
   INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK TO my_point;
END;
```

The number of active savepoints per session is unlimited. An *active savepoint* is one marked since the last commit or rollback.

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. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to OUT parameters. Also, PL/SQL does not roll back database work done by the subprogram.

Ending Transactions

A good programming practice is to commit or roll back every transaction explicitly. Whether you issue the commit or rollback in your PL/SQL program or in the host environment depends on the flow of application logic. If you neglect to commit or roll back a transaction explicitly, the host environment determines its final state.

For example, in the SQL*Plus environment, if your PL/SQL block does not include a COMMIT or ROLLBACK statement, the final state of your transaction depends on what you do after running the block. If you execute a data definition, data control, or COMMIT statement or if you issue the EXIT, DISCONNECT, or QUIT command, Oracle commits the transaction. If you execute a ROLLBACK statement or abort the SQL*Plus session, Oracle rolls back the transaction.

In the Oracle Precompiler environment, if your program does not terminate normally, Oracle rolls back your transaction. A program terminates normally when it explicitly commits or rolls back work and disconnects from Oracle using the RELEASE parameter, as follows:

```
EXEC SQL COMMIT WORK RELEASE;
```

Using SET TRANSACTION

You use the SET TRANSACTION statement to begin a read-only or read-write transaction, establish an isolation level, or assign your current transaction to a specified rollback segment. Read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read-consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction. In the example below, as a store manager, you use a read-only transaction to gather sales figures for the day, the past week, and the past month. The figures are unaffected by other users updating the database during the transaction.

```
DECLARE
   daily_sales   REAL;
   weekly_sales  REAL;
   monthly_sales REAL;
BEGIN
   ...
   COMMIT;  -- ends previous transaction
   SET TRANSACTION READ ONLY;
   SELECT SUM(amt) INTO daily_sales FROM sales
      WHERE dte = SYSDATE;
   SELECT SUM(amt) INTO weekly_sales FROM sales
      WHERE dte > SYSDATE - 7;
   SELECT SUM(amt) INTO monthly_sales FROM sales
      WHERE dte > SYSDATE - 30;
   COMMIT;  -- ends read-only transaction
   ...
END;
```

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can only appear once in a transaction. If you set a transaction to READ ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

Restrictions on SET TRANSACTION

Only the SELECT INTO, OPEN, FETCH, CLOSE, LOCK TABLE, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. Also, queries cannot be FOR UPDATE.

Overriding Default Locking

By default, Oracle locks data structures for you automatically. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction.

With the LOCK TABLE statement, you can explicitly lock entire tables. With the SELECT FOR UPDATE statement, you can explicitly lock specific rows of a table to make sure they do not change before an update or delete is executed. However, Oracle automatically obtains row-level locks at update or delete time. So, use the FOR UPDATE clause only if you want to lock the rows *before* the update or delete.

Using FOR UPDATE

When you declare a cursor that will be referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks. An example follows:

```
DECLARE
   CURSOR c1 IS SELECT empno, sal FROM emp
      WHERE job = 'SALESMAN' AND comm > sal
      FOR UPDATE NOWAIT;
```

The FOR UPDATE clause 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 the table has 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 table is available.

All rows are locked when you open the cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. So, you cannot fetch from a FOR UPDATE cursor after a commit. (For a workaround, see "Fetching Across Commits" on page 5-49.)

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. For example, the statement below locks the emp table in *row share* mode. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use. Table locks are released when your transaction issues a commit or rollback.

```
LOCK TABLE emp IN ROW SHARE MODE NOWAIT;
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an *exclusive* lock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table. For more information about lock modes, see *Oracle8i Application Developer's Guide - Fundamentals*.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row will one transaction wait for the other to complete.

Fetching Across Commits

The FOR UPDATE clause acquires exclusive row locks. All rows are locked when you open the cursor, and they are unlocked when you commit your transaction. So, you cannot fetch from a FOR UPDATE cursor after a commit. If you do, PL/SQL raises an exception. In the following example, the cursor FOR loop fails after the tenth insert:

```
DECLARE
   CURSOR c1 IS SELECT ename FROM emp FOR UPDATE OF sal;
   ctr NUMBER := 0;
BEGIN
   FOR emp_rec IN c1 LOOP  -- FETCHes implicitly
      ...
      ctr := ctr + 1;
      INSERT INTO temp VALUES (ctr, 'still going');
      IF ctr >= 10 THEN
         COMMIT;  -- releases locks
      END IF;
   END LOOP;
END;
```

If you want to fetch across commits, do not use the FOR UPDATE and CURRENT OF clauses. Instead, use the ROWID pseudocolumn to mimic the CURRENT OF clause. Simply select the rowid of each row into a UROWID variable. Then, use the rowid to identify the current row during subsequent updates and deletes. An example follows:

```
DECLARE
   CURSOR c1 IS SELECT ename, job, rowid FROM emp;
   my_ename  emp.ename%TYPE;
   my_job    emp.job%TYPE;
   my_rowid  UROWID;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_ename, my_job, my_rowid;
      EXIT WHEN c1%NOTFOUND;
      UPDATE emp SET sal = sal * 1.05 WHERE rowid = my_rowid;
      -- this mimics WHERE CURRENT OF c1
      COMMIT;
   END LOOP;
   CLOSE c1;
END;
```

Be careful. In the last example, the fetched rows are *not* locked because no FOR UPDATE clause is used. So, other users might unintentionally overwrite your changes. Also, the cursor must have a read-consistent view of the data, so rollback segments used in the update are not released until the cursor is closed. This can slow down processing when many rows are updated.

The next example shows that you can use the %ROWTYPE attribute with cursors that reference the ROWID pseudocolumn:

```
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;
```

Dealing with Size Limitations

PL/SQL was designed for high-speed transaction processing. As a result, the compiler limits the number of tokens (identifiers, keywords, operators, and so on) that a program unit (block, subprogram, or package) can contain. Units that exceed the limit cause a *program too large* compilation error. Generally, unit specs larger than 32K and unit bodies larger than 64K exceed the token limit. However, smaller units can also exceed the limit if they contain many variables or complex SQL statements.

Typically, this problem occurs with package bodies or anonymous blocks. With a package, the best solution is to divide it into smaller packages. With a block, the best solution is to redefine it as a series of subprograms, which can be stored in the database. For more information, see Chapter 7, "Subprograms".

Another solution is to break the block into two sub-blocks. Consider the SQL*Plus script below. Before the first block terminates, it inserts any data the second block needs into a database table called temp. When the second block starts executing, it selects the data from temp. This approximates the passing of parameters from one procedure to another.

```
DECLARE
   mode   NUMBER;
   median NUMBER;
BEGIN
   ...
   INSERT INTO temp (col1, col2, col3)
      VALUES (mode, median, 'blockA');
END;
/
DECLARE
   mode   NUMBER;
   median NUMBER;
BEGIN
   SELECT col1, col2 INTO mode, median FROM temp
      WHERE col3 = 'blockA';
   ...
END;
/
```

This method works unless you must re-execute the first block while the second block is still executing, or unless two or more users must run the script concurrently.

Alternatively, you can embed the blocks in a host language such as C or COBOL. That way, you can re-execute the first block using flow-of-control statements. Also, you can store data in global host variables instead of a database table. For example, you might embed the following two blocks in a Pro*C program:

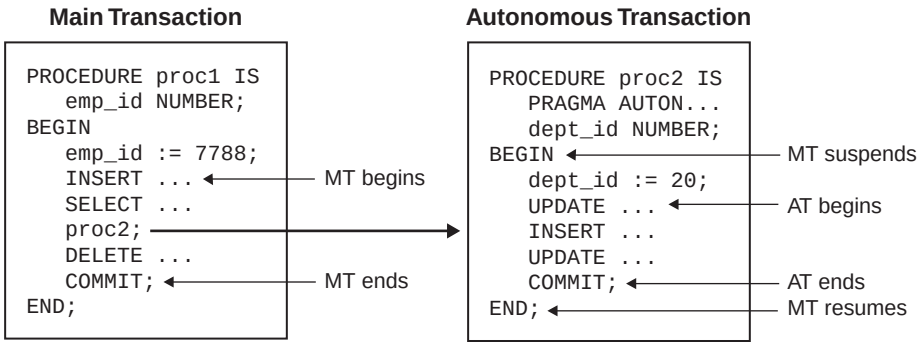
```
/* The host variables 'my_sal', 'my_comm', and 'my_empno'
   are assigned values in the host environment. The host
   variable 'comm_ind' is an indicator variable. */
BEGIN
   SELECT sal, comm INTO :my_sal, :my_comm:comm_ind FROM emp
      WHERE empno = :my_empno;
   IF :my_comm:comm_ind IS NULL THEN
      ...
   END IF;
END;

BEGIN
   ...
   IF :my_comm:comm_ind > 1000 THEN
      :my_sal := :my_sal * 1.10;
      UPDATE emp SET sal = :my_sal WHERE empno = :my_empno;
   END IF;
END;
```

Using Autonomous Transactions

A transaction is a series of SQL statements that does a logical unit of work. Often, one transaction starts another. In some applications, a transaction must operate outside the scope of the transaction that started it. This can happen, for example, when a transaction calls out to a data cartridge.

An *autonomous transaction* is an independent transaction started by another transaction, the *main transaction*. Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction. Figure 5–1 shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

Figure 5–1 Transaction Control Flow



Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. So, you can log events, increment retry counters, and so on, even if the main transaction rolls back.

More important, autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure's autonomous operations, and the procedure need not know about the application's transaction context. That makes autonomous transactions less error-prone than regular transactions and easier to use.

Furthermore, autonomous transactions have all the functionality of regular transactions. They allow parallel queries, distributed processing, and all the transaction control statements including `SET TRANSACTION`.

Defining Autonomous Transactions

To define autonomous transactions, you use the pragma (compiler directive) AUTONOMOUS_TRANSACTION. The pragma instructs the PL/SQL compiler to mark a routine as *autonomous* (independent). In this context, the term *routine* includes

- top-level (not nested) anonymous PL/SQL blocks
- local, stand-alone, and packaged functions and procedures
- methods of a SQL object type
- database triggers

You can code the pragma anywhere in the declarative section of a routine. But, for readability, code the pragma at the top of the section. The syntax follows:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

In the following example, you mark a packaged function as autonomous:

```
CREATE PACKAGE banking AS
   ...
   FUNCTION balance (acct_id INTEGER) RETURN REAL;
END banking;

CREATE PACKAGE BODY banking AS
   ...
   FUNCTION balance (acct_id INTEGER) RETURN REAL IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      my_bal REAL;
   BEGIN
      ...
   END;
END banking;
```

Restriction: You cannot use the pragma to mark all subprograms in a package (or all methods in an object type) as autonomous. Only individual routines can be marked autonomous. For example, the following pragma is illegal:

```
CREATE PACKAGE banking AS
   PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
   ...
   FUNCTION balance (acct_id INTEGER) RETURN REAL;
END banking;
```

In the next example, you mark a stand-alone procedure as autonomous:

```
CREATE PROCEDURE close_account (acct_id INTEGER, OUT balance) AS
   PRAGMA AUTONOMOUS_TRANSACTION;
   my_bal REAL;
BEGIN
   ...
END;
```

In the following example, you mark a PL/SQL block as autonomous:

```
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
   my_empno NUMBER(4);
BEGIN
   ...
END;
```

Restriction: You cannot mark a nested PL/SQL block as autonomous.

In the example below, you mark a database trigger as autonomous. Unlike regular triggers, autonomous triggers can contain transaction control statements such as COMMIT and ROLLBACK.

```
CREATE TRIGGER parts_trigger
BEFORE INSERT ON parts FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO parts_log VALUES(:new.pnum, :new.pname);
   COMMIT;  -- allowed only in autonomous triggers
END;
```

Autonomous versus Nested Transactions

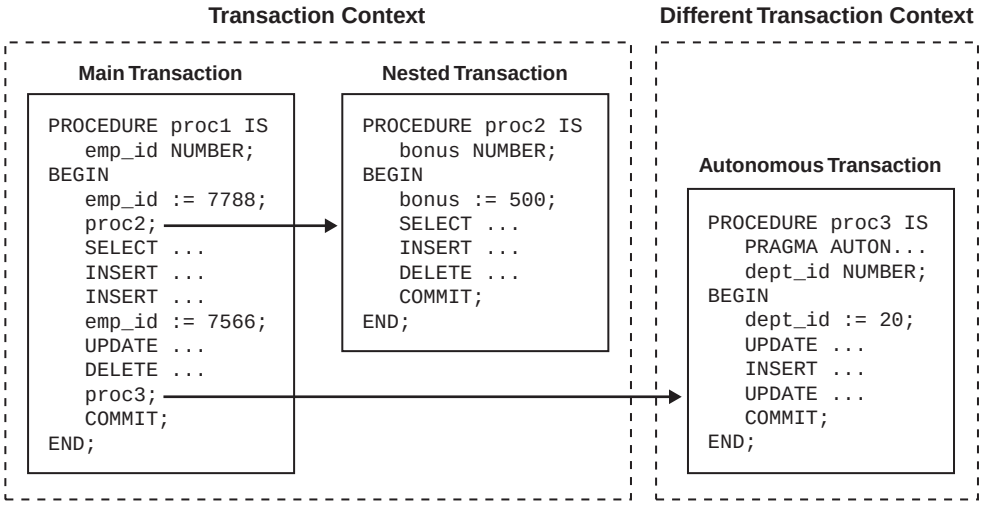
Although an autonomous transaction is started by another transaction, it is *not* a nested transaction for the following reasons:

- It does not share transactional resources (such as locks) with the main transaction.
- It does not depend on the main transaction. For example, if the main transaction rolls back, nested transactions roll back, but autonomous transactions do not.
- Its committed changes are visible to other transactions immediately. (A nested transaction's committed changes are not visible to other transactions until the main transaction commits.)

Transaction context

As Figure 5–2 shows, the main transaction shares its context with nested transactions, but not with autonomous transactions. Likewise, when one autonomous routine calls another (or itself recursively), the routines share no transaction context. However, when an autonomous routine calls a non-autonomous routine, the routines share the same transaction context.

Figure 5–2 Transaction Context



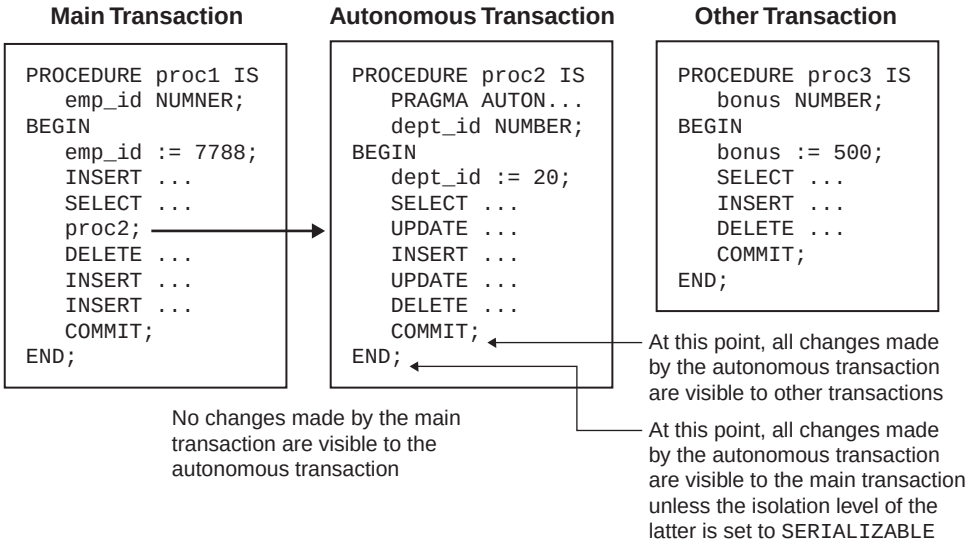
Transaction Visibility

As Figure 5–3 on page 5-56 shows, changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. The changes also become visible to the main transaction when it resumes, but only if its isolation level is set to READ COMMITTED (the default).

If you set the isolation level of the main transaction to SERIALIZABLE, as follows, changes made by its autonomous transactions are *not* visible to the main transaction when it resumes:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Figure 5–3 Transaction Visibility



Controlling Autonomous Transactions

The first SQL statement in an autonomous routine begins a transaction. When one transaction ends, the next SQL statement begins another transaction. All SQL statements executed since the last commit or rollback make up the current transaction. To control autonomous transactions, use the following statements, which apply only to the current (active) transaction:

- `COMMIT`
- `ROLLBACK [TO savepoint_name]`
- `SAVEPOINT savepoint_name`
- `SET TRANSACTION`

`COMMIT` ends the current transaction and makes permanent changes made during that transaction. `ROLLBACK` ends the current transaction and undoes changes made during that transaction. `ROLLBACK TO` undoes part of a transaction. `SAVEPOINT` names and marks the current point in a transaction. `SET TRANSACTION` sets transaction properties such as read / write access and isolation level.

Note: Transaction properties set in the main transaction apply only to that transaction, not to its autonomous transactions, and vice versa.

Entering and Exiting

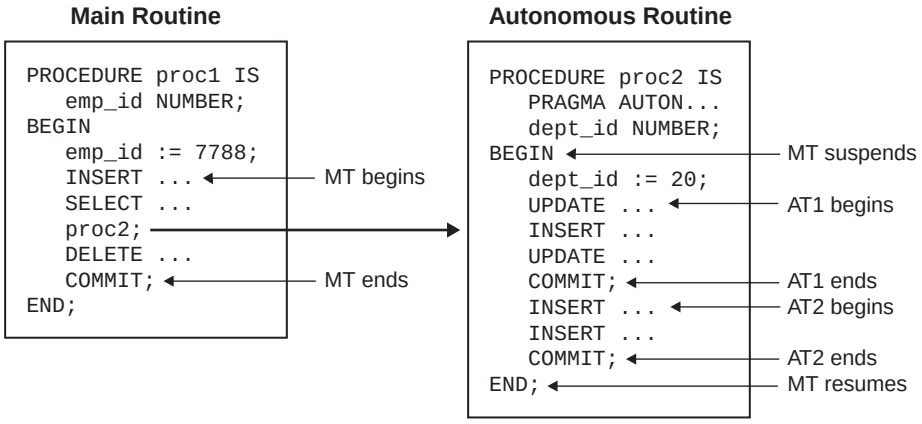
When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes.

To exit normally, you must explicitly commit or roll back all autonomous transactions. If the routine (or any routine called by it) has pending transactions, an exception is raised, and the pending transactions are rolled back.

Committing and Rolling Back

COMMIT and ROLLBACK end the active autonomous transaction but do not exit the autonomous routine. As Figure 5–4 shows, when one transaction ends, the next SQL statement begins another transaction.

Figure 5–4 Multiple Autonomous Transactions



Using Savepoints

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. So, when in an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

Avoiding Errors

To avoid some common errors, keep the following points in mind when designing autonomous transactions:

- If an autonomous transaction attempts to access a resource held by the main transaction (which cannot resume until the autonomous routine exits), a deadlock can occur.
- The Oracle initialization parameter TRANSACTIONS specifies the maximum number of concurrent transactions. That number might be exceeded if autonomous transactions (which run concurrently with the main transaction) are not taken into account.

Example 1: Using an Autonomous Trigger

Among other things, you can use database triggers to log events transparently. Suppose you want to track all inserts into a table, even those that roll back. In the example below, you use a trigger to insert duplicate rows into a shadow table. Because it is autonomous, the trigger can commit inserts into the shadow table whether or not you commit inserts into the main table.

```
-- 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));

-- ceate 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
```

Example 2: Calling an Autonomous Function from SQL

A function called from SQL statements must obey certain rules meant to control side effects. (See "Controlling Sides Effects" on page 7-7.) To check for violations of the rules, you can use the pragma `RESTRICT_REFERENCES`, which instructs the compiler to report reads of / writes to database tables, package variables, or both. (See *Oracle8i Application Developer's Guide - Fundamentals*.)

However, all autonomous routines have read / write access to the database. So, they never violate the rules "read no database state" and "write no database state." 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 BODY debugging AS
   FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
      PRAGMA AUTONOMOUS_TRANSACTION;
```

```
   BEGIN
      -- the following insert does not violate the constraint
      -- WNDS because this is an autonomous routine
      INSERT INTO debug_output VALUES (msg);
      COMMIT;
      RETURN msg;
   END;
END debugging;

-- call the packaged function from a query
DECLARE
   my_empno NUMBER(4);
   my_ename VARCHAR2(15);
BEGIN
   ...
   SELECT debugging.log_msg(ename) INTO my_ename FROM emp
      WHERE empno = my_empno;
   -- 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;
```

Improving Performance

This section gives several techniques for improving performance and explains how your applications can use them.

Use Object Types and Collections

Collection types (see Chapter 4) and object types (see Chapter 9) increase your productivity by allowing for realistic data modeling. Complex real-world entities and relationships map directly into object types. And, a well-constructed object model can improve application performance by eliminating table joins, reducing round trips, and the like.

Client programs, including PL/SQL programs, can declare objects and collections, pass them as parameters, store them in the database, retrieve them, and so on. Also, by encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods.

Objects and collections are more efficient to store and retrieve because they can be manipulated as a whole. Also, object support is integrated architecturally with the database, so it can take advantage of the many scalability and performance improvements built into Oracle8*i*.

Use Bulk Binds

When SQL statements execute inside a loop using collection elements as bind variables, context switching between the PL/SQL and SQL engines can slow down execution. For example, the following UPDATE statement is sent to the SQL engine with each iteration of the FOR loop:

```
DECLARE
   TYPE NumList IS VARRAY(20) OF NUMBER;
   depts NumList := NumList(10, 30, 70, ...);  -- department numbers
BEGIN
   ...
   FOR i IN depts.FIRST..depts.LAST LOOP
      ...
      UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
   END LOOP;
END;
```

In such cases, if the SQL statement affects five or more database rows, the use of bulk binds can improve performance considerably. For example, the following UPDATE statement is sent to the SQL engine just once, with the entire nested table:

```
FORALL i IN depts.FIRST..depts.LAST
   UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
```

To maximize performance, rewrite your programs as follows:

- If an INSERT, UPDATE, or DELETE statement executes inside a loop and references collection elements, move it into a FORALL statement.
- If a SELECT INTO, FETCH INTO, or RETURNING INTO clause references a collection, incorporate the BULK COLLECT clause.
- If possible, use host arrays to pass collections back and forth between your programs and the database server.

Note: These are not a trivial tasks. They require careful analysis of program control-flows and dependencies. For more information about bulk binding, see "Taking Advantage of Bulk Binds" on page 4-29.

Use Native Dynamic SQL

Some programs (a general-purpose report writer for example) must build and process a variety of SQL statements at run time. So, their full text is unknown until then. Such statements can, and probably will, change from execution to execution. So, they are called *dynamic* SQL statements.

Formerly, to execute dynamic SQL statements, you had to use the supplied package DBMS_SQL. Now, within PL/SQL, you can execute any kind of dynamic SQL statement using an interface called *native dynamic SQL*.

Native dynamic SQL is easier to use and much faster than package DBMS_SQL. In the following example, you declare a cursor variable, then associate it with a dynamic SELECT statement that returns rows from database table emp:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv   EmpCurTyp;
   my_ename VARCHAR2(15);
   my_sal   NUMBER := 1000;
BEGIN
   OPEN emp_cv FOR
      'SELECT ename, sal FROM emp
          WHERE sal > :s' USING my_sal;
   ...
END;
```

For more information, see Chapter 10, "Native Dynamic SQL".

Use External Routines

PL/SQL is specialized for SQL transaction processing. So, some tasks are more quickly done in a lower-level language such as C, which is very efficient at machine-precision calculations.

PL/SQL extends the functionality of the Oracle server by providing an interface for calling routines written in other languages. Standard libraries already written and available in other languages can be called from PL/SQL programs. This promotes reusability, efficiency, and modularity.

To speed up execution, you can rewrite computation-bound programs in C. In addition, you can move such programs from client to server, where they will execute faster thanks to more computing power and less across-network communication.

For example, you can write methods for an image object type in C, store them in a dynamic link library (DLL), register the library with PL/SQL, then call it from your applications. At run time, the library loads dynamically and, for safety, runs in a separate address space (implemented as a separate process).

For more information, see *Oracle8i Application Developer's Guide - Fundamentals*.

Use the NOCOPY Compiler Hint

By default, OUT and IN OUT parameters are passed by value. That is, the value of an IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory. To prevent that, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. In the following example, you ask the compiler to pass IN OUT parameter my_unit by reference instead of by value:

```
DECLARE
   TYPE Platoon IS VARRAY(200) OF Soldier;
   PROCEDURE reorganize (my_unit IN OUT NOCOPY Platoon) IS ...
BEGIN
   ...
END;
```

For more information, see "NOCOPY Compiler Hint" on page 7-17.

Use the RETURNING Clause

Often, applications need information about the row affected by a SQL operation, for example, to generate a report or take a subsequent action. The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to SELECT the row after an insert or update, or before a delete. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

In the following example, you update the salary of an employee and at the same time retrieve the employee's name and new salary into PL/SQL variables.

```
PROCEDURE update_salary (emp_id NUMBER) IS
   name    VARCHAR2(15);
   new_sal NUMBER;
BEGIN
   UPDATE emp SET sal = sal * 1.1
      WHERE empno = emp_id
      RETURNING ename, sal INTO name, new_sal;
```

Use Serially Reusable Packages

To help you manage the use of memory, PL/SQL provides the pragma SERIALLY_REUSABLE, which lets you mark some packages as *serially reusable*. You can so mark a package if its state is needed only for the duration of one call to the server (for example, an OCI call to the server or a server-to-server RPC).

The global memory for such packages is pooled in the System Global Area (SGA), not allocated to individual users in the User Global Area (UGA). That way, the package work area can be reused. When the call to the server ends, the memory is returned to the pool. Each time the package is reused, its public variables are initialized to their default values or to NULL.

The maximum number of work areas needed for a package is the number of concurrent users of that package, which is usually much smaller than the number of logged-on users. The increased use of SGA memory is more than offset by the decreased use of UGA memory. Also, Oracle ages-out work areas not in use if it needs to reclaim SGA memory.

For packages without a body, you code the pragma in the package spec using the following syntax:

```
PRAGMA SERIALLY_REUSABLE;
```

For packages with a body, you must code the pragma in the spec and body. You cannot code the pragma only in the body. The following example shows how a public variable in a serially reusable package behaves across call boundaries:

```
CREATE OR REPLACE PACKAGE sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   num NUMBER := 0;
   PROCEDURE init_pkg_state(n NUMBER);
   PROCEDURE print_pkg_state;
END sr_pkg;
/
```

```
CREATE OR REPLACE PACKAGE BODY sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   /* Initialize package state. */
   PROCEDURE init_pkg_state (n NUMBER) IS
   BEGIN
      sr_pkg.num := n;
   END;
   /* Print package state. */
   PROCEDURE print_pkg_state IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('Num is: ' || sr_pkg.num);
   END;
END sr_pkg;
/
BEGIN
   /* Initialize package state. */
   sr_pkg.init_pkg_state(4);
   /* On same server call, print package state. */
   sr_pkg.print_pkg_state;  -- prints 4
END;
/
-- subsequent server call
BEGIN
   -- package's public variable will be initialized to its
   -- default value automatically
   sr_pkg.print_pkg_state;  -- prints 0
END;
```

For more information, see *Oracle8i Supplied Packages Reference.*

Use the PLS_INTEGER Datatype

When you need to declare an integer variable, use the datatype PLS_INTEGER, which is the most efficient numeric type. That is because PLS_INTEGER values require less storage than INTEGER or NUMBER values, which are represented internally as 22-byte Oracle numbers. Also, PLS_INTEGER operations use machine arithmetic, so they are faster than BINARY_INTEGER, INTEGER, or NUMBER operations, which use library arithmetic.

Furthermore, INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNTYPE are constrained subtypes. So, their variables require precision checking at run time, which can affect performance.

Avoid the NOT NULL Constraint

In PL/SQL, using the NOT NULL constraint incurs a performance cost. Consider the following example:

```
PROCEDURE calc_m IS
   m NUMBER NOT NULL;
   a NUMBER;
   b NUMBER;
BEGIN
   ...
   m := a + b;
```

Because m is constrained by NOT NULL, the value of the expression a + b is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to m. Otherwise, an exception is raised. However, if m were not constrained, the value would be assigned to m directly.

A more efficient way to write the last example follows:

```
PROCEDURE calc_m IS
   m NUMBER;  -- no constraint
   a NUMBER;
   b NUMBER;
BEGIN
   ...
   m := a + b;
   IF m IS NULL THEN ...  -- enforce constraint programmatically
END;
```

Note that the subtypes NATURALN and POSTIVEN are defined as NOT NULL. So, using them incurs the same performance cost.

Rephrase Conditional Control Statements

When evaluating a logical expression, PL/SQL uses short-circuit evaluation. That is, PL/SQL stops evaluating the expression as soon as the result can be determined. For example, in the following OR expression, when the value of sal is less than 1500, the left operand yields TRUE, so PL/SQL need not evaluate the right operand (because OR returns TRUE if either of its operands is true):

```
IF (sal < 1500) OR (comm IS NULL) THEN
   ...
END IF;
```

Now, consider the following AND expression:

```
IF credit_ok(cust_id) AND (loan < 5000) THEN
   ...
END IF;
```

The Boolean function credit_ok is always called. However, if you switch the operands of AND as follows

```
IF (loan < 5000) AND credit_ok(cust_id) THEN
   ...
END IF;
```

the function is called only when the expression loan < 5000 is true (because AND returns TRUE only if both its operands are true).

The same idea applies to EXIT-WHEN statements.

Avoid Implicit Datatype Conversions

At run time, PL/SQL converts between structurally different datatypes implicitly. For instance, assigning a PLS_INTEGER variable to a NUMBER variable results in a conversion because their internal representations are different.

Avoiding implicit conversions can improve performance. Look at the example below. The integer literal 15 is represented internally as a signed 4-byte quantity, so PL/SQL must convert it to an Oracle number before the addition. However, the floating-point literal 15.0 is represented as a 22-byte Oracle number, so no conversion is necessary.

```
DECLARE
   n NUMBER;
   c CHAR(5);
BEGIN
   n := n + 15;    -- converted
   n := n + 15.0;  -- not converted
```

Here is another example:

```
DECLARE
   c CHAR(5);
BEGIN
   c := 25;    -- converted
   c := '25';  -- not converted
```

Ensuring Backward Compatibility

PL/SQL Version 2 allows some abnormal behavior that Version 8 disallows. Specifically, Version 2 allows you to

- make forward references to RECORD and TABLE types when declaring variables
- specify the name of a variable (not a datatype) in the RETURN clause of a function spec
- assign values to the elements of an index-by table IN parameter
- pass the fields of a record IN parameter to another subprogram as OUT parameters
- use the fields of a record OUT parameter on the right-hand side of an assignment statement
- use OUT parameters in the FROM list of a SELECT statement

For backward compatibility, you might want to keep this particular Version 2 behavior. You can do that by setting the PLSQL_V2_COMPATIBILITY flag. On the server side, you can set the flag in two ways:

- Add the following line to the Oracle initialization file:

```
PLSQL_V2_COMPATIBILITY=TRUE
```

- Execute one of the following SQL statements:

```
ALTER SESSION SET PLSQL_V2_COMPATIBILITY = TRUE;
ALTER SYSTEM SET PLSQL_V2_COMPATIBILITY = TRUE;
```

If you specify FALSE (the default), only Version 8 behavior is allowed.

On the client side, a command-line option sets the flag. For example, in the Oracle Precompilers environment, you specify the runtime option DBMS on the command line, as follows:

```
... DBMS=V7 ...
```

6

Error Handling

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.

Major Topics

Overview

In PL/SQL, a warning or error condition is called an *exception*. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include *division by zero* and *out of memory*. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named insufficient_funds to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions *must* be given names.

When an error occurs, an exception is *raised*. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called *exception handlers*. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

In the example below, you calculate and store a price-to-earnings ratio for a company with ticker symbol XYZ. If the company has zero earnings, the predefined exception ZERO_DIVIDE is raised. This stops normal execution of the block and transfers control to the exception handlers. The optional OTHERS handler catches all exceptions that the block does not name specifically.

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   SELECT price / earnings INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';  -- might cause division-by-zero error
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
   COMMIT;
EXCEPTION  -- exception handlers begin
   WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
      COMMIT;
   ...
   WHEN OTHERS THEN  -- handles all other errors
      ROLLBACK;
END;  -- exception handlers and block end here
```

The last example illustrates exception handling, not the effective use of INSERT statements. For example, a better way to do the insert follows:

```
INSERT INTO stats (symbol, ratio)
   SELECT symbol, DECODE(earnings, 0, NULL, price / earnings)
   FROM stocks WHERE symbol = 'XYZ';
```

In this example, a subquery supplies values to the INSERT statement. If earnings are zero, the function DECODE returns a null. Otherwise, DECODE returns the price-to-earnings ratio.

Advantages of Exceptions

Using exceptions for error handling has several advantages. Without exception handling, every time you issue a command, you must check for execution errors:

```
BEGIN
   SELECT ...
      -- check for 'no data found' error
   SELECT ...
      -- check for 'no data found' error
   SELECT ...
      -- check for 'no data found' error
```

Error processing is not clearly separated from normal processing; nor is it robust. If you neglect to code a check, the error goes undetected and is likely to cause other, seemingly unrelated errors.

With exceptions, you can handle errors conveniently without the need to code multiple checks, as follows:

```
BEGIN
   SELECT ...
   SELECT ...
   SELECT ...
   ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
```

Exceptions improve readability by letting you isolate error-handling routines. The primary algorithm is not obscured by error recovery algorithms. Exceptions also improve reliability. You need not worry about checking for an error at every point it might occur. Just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.

Predefined Exceptions

An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.

To handle other Oracle errors, you can use the OTHERS handler. The functions SQLCODE and SQLERRM are especially useful in the OTHERS handler because they return the Oracle error code and message text. Alternatively, you can use the pragma EXCEPTION_INIT to associate exception names with Oracle error codes.

PL/SQL declares predefined exceptions globally in package STANDARD, which defines the PL/SQL environment. So, you need not declare them yourself. You can write handlers for predefined exceptions using the names shown in the list below. Also shown are the corresponding Oracle error codes and SQLCODE return values.

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
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
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	Your program attempts to assign values to the attributes of an uninitialized (atomically null) object.
COLLECTION_IS_NULL	Your program attempts to apply collection methods other than EXISTS to an uninitialized (atomically null) nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	Your program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers. So, your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	Your program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	Your program attempts an illegal cursor operation 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.)
LOGIN_DENIED	Your program attempts to log on to Oracle with an invalid username and/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. SQL aggregate functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls a aggregate function will never raise NO_DATA_FOUND. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.
NOT_LOGGED_ON	Your program issues a database call without being connected to Oracle.
PROGRAM_ERROR	PL/SQL has an internal problem.

Exception	Raised when ...
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	Your program attempts to call a MEMBER method on a null instance. That is, the built-in parameter SELF (which is always the first parameter passed to a MEMBER method) is null.
STORAGE_ERROR	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	Your program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	Your 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	Your program attempts to divide a number by zero.

User-Defined 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 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

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. So, the sub-block cannot reference the global exception unless it was declared in a labeled block, in which case the following syntax is valid:

```
block_label.exception_name
```

The following example illustrates the scope rules:

```
DECLARE
   past_due EXCEPTION;
   acct_num NUMBER;
BEGIN
   DECLARE  ---------- sub-block begins
      past_due EXCEPTION;  -- this declaration prevails
      acct_num NUMBER;
```

```
   BEGIN
      ...
      IF ... THEN
         RAISE past_due;  -- this is not handled
      END IF;
   END;  ------------- sub-block ends
EXCEPTION
   WHEN past_due THEN  -- does not handle RAISEd exception
      ...
END;
```

The enclosing block does not handle the raised exception because the declaration of past_due in the sub-block prevails. Though they share the same name, the two past_due exceptions are different, just as the two acct_num variables share the same name but are different variables. Therefore, the RAISE statement and the WHEN clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the sub-block or define an OTHERS handler.

Using EXCEPTION_INIT

To handle unnamed internal exceptions, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A *pragma* is a compiler directive, which can be thought of as a parenthetical remark to the compiler. Pragmas (also called *pseudoinstructions*) are processed at compile time, not at run time. For example, in the language Ada, the following pragma tells the compiler to optimize the use of storage space:

```
pragma OPTIMIZE(SPACE);
```

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, Oracle_error_number);
```

where exception_name is the name of a previously declared exception. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:

```
DECLARE
   deadlock_detected EXCEPTION;
   PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
```

```
BEGIN
   ...
EXCEPTION
   WHEN deadlock_detected THEN
      -- handle the error
END;
```

Using raise_application_error

Package DBMS_STANDARD, which is supplied with Oracle, provides language facilities that help your application interact with Oracle. For example, the procedure `raise_application_error` lets you issue user-defined error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To call `raise_application_error`, use the syntax

```
raise_application_error(error_number, message[, {TRUE | FALSE}]);
```

where `error_number` is a negative integer in the range -20000 .. -20999 and `message` is a character string up to 2048 bytes long. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors. Package DBMS_STANDARD is an extension of package STANDARD, so you need not qualify references to its contents.

An application can call `raise_application_error` only from an executing stored subprogram (or method). When called, `raise_application_error` ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

In the following example, you call `raise_application_error` if an employee's salary is missing:

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) AS
   curr_sal NUMBER;
BEGIN
   SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
   IF curr_sal IS NULL THEN
      /* Issue user-defined error message. */
      raise_application_error(-20101, 'Salary is missing');
   ELSE
      UPDATE emp SET sal = curr_sal + amount WHERE empno = emp_id;
   END IF;
END raise_salary;
```

The calling application gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS handler. Also, it can use the pragma EXCEPTION_INIT to map specific error numbers returned by raise_application_error to exceptions of its own, as follows:

```
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 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.

Using 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(4);
BEGIN
   ...
   IF number_on_hand < 1 THEN
      RAISE out_of_stock;
   END IF;
EXCEPTION
   WHEN out_of_stock THEN
      -- handle the error
END;
```

You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as the following example shows:

```
DECLARE
   acct_type INTEGER;
BEGIN
   ...
   IF acct_type NOT IN (1, 2, 3) THEN
      RAISE INVALID_NUMBER;  -- raise predefined exception
   END IF;
EXCEPTION
   WHEN INVALID_NUMBER THEN
      ROLLBACK;
   ...
END;
```

How Exceptions Propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception *propagates*. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. In the latter case, PL/SQL returns an *unhandled exception* error to the host environment.

However, exceptions cannot propagate across remote procedure calls (RPCs). Therefore, a PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see "Using raise_application_error" on page 6-9.

Figure 6–1, Figure 6–2, and Figure 6–3 illustrate the basic propagation rules.

Figure 6–1 Propagation Rules: Example 1

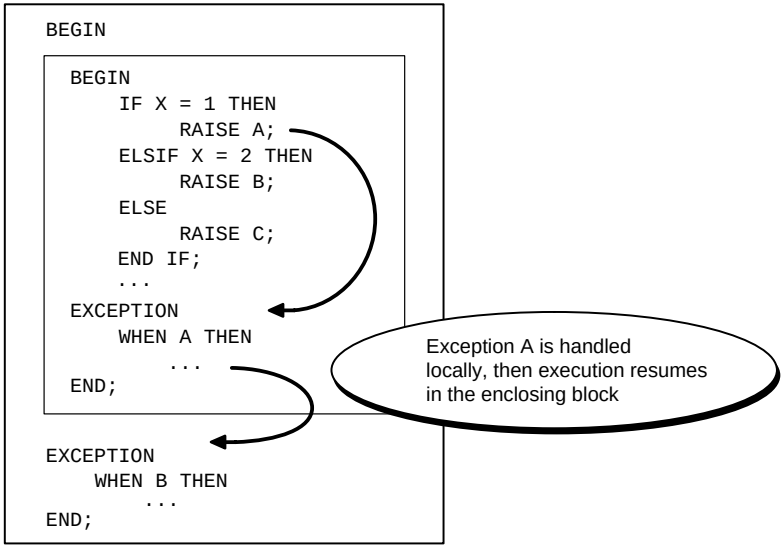


Figure 6–2 Propagation Rules: Example 2

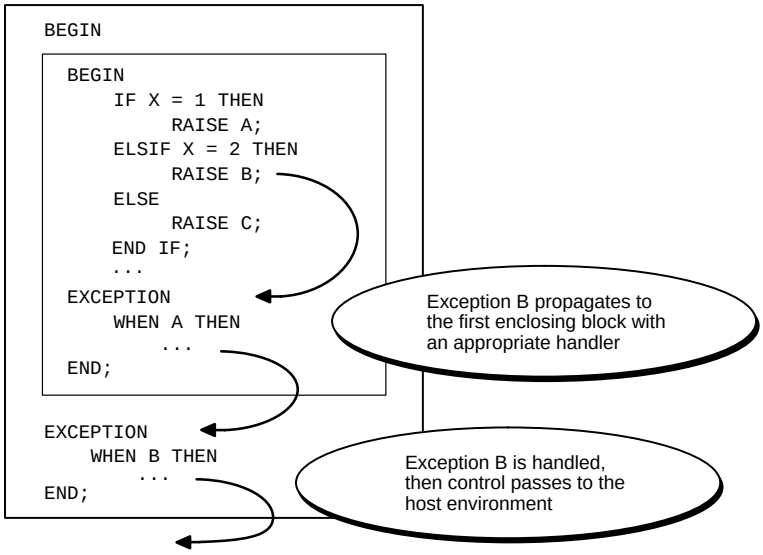
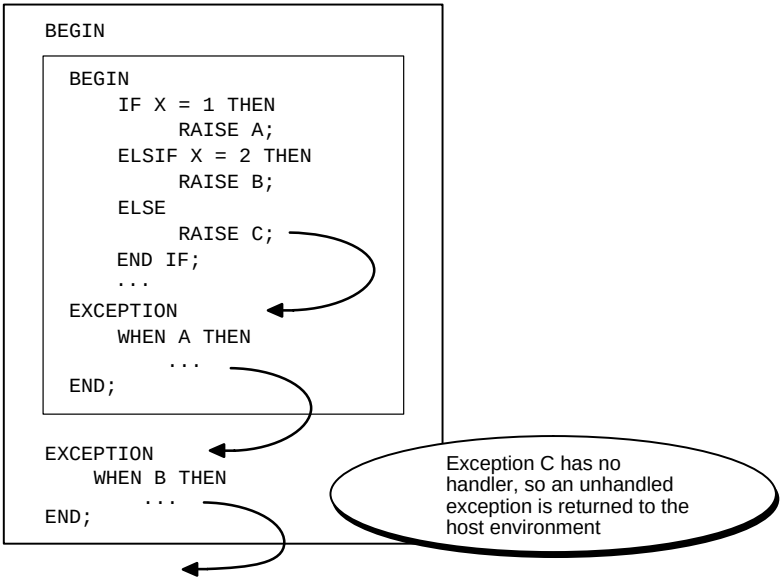


Figure 6–3 Propagation Rules: Example 3



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;
   BEGIN
      ...
      IF ... THEN
         RAISE past_due;
      END IF;
   END;  ------------- sub-block ends
EXCEPTION
   ...
   WHEN OTHERS THEN
      ROLLBACK;
END;
```

Because the block in which exception `past_due` was declared has no handler for it, the exception propagates to the enclosing block. But, according to the scope rules, enclosing blocks cannot reference exceptions declared in a sub-block. So, only an `OTHERS` handler can catch the exception.

Reraising an Exception

Sometimes, you want to *reraise* an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, simply place a `RAISE` statement in the local handler, as shown in the following example:

```
DECLARE
   out_of_balance  EXCEPTION;
BEGIN
   ...
   BEGIN  ---------- sub-block begins
      ...
      IF ... THEN
         RAISE out_of_balance;  -- raise the exception
      END IF;
```

```
   EXCEPTION
      WHEN out_of_balance THEN
         -- handle the error
         RAISE;  -- reraise the current exception
   END;  ------------ sub-block ends
EXCEPTION
   WHEN out_of_balance THEN
      -- handle the error differently
   ...
END;
```

Omitting the exception name in a RAISE statement—allowed only in an exception handler—reraises the current exception.

Handling Raised 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.

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
   ...
```

```
EXCEPTION
   WHEN OTHERS THEN -- cannot catch the exception
      ...
END;
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates *immediately* to the enclosing block.

Exceptions Raised in Handlers

Only one exception at a time can be active in the exception-handling part of a block or subprogram. So, an exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for the newly raised exception. From there on, the exception propagates normally. Consider the following example:

```
EXCEPTION
   WHEN INVALID_NUMBER THEN
      INSERT INTO ...  -- might raise DUP_VAL_ON_INDEX
   WHEN DUP_VAL_ON_INDEX THEN ...  -- cannot catch the exception
END;
```

Branching to or from an Exception Handler

A GOTO statement cannot branch into an exception handler. Also, a GOTO statement cannot branch from an exception handler into the current block. For example, the following GOTO statement is illegal:

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   DELETE FROM stats WHERE symbol = 'XYZ';
   SELECT price / NVL(earnings, 0) INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';
   <<my_label>>
   INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      pe_ratio := 0;
      GOTO my_label;  -- illegal branch into current block
END;
```

However, a GOTO statement can branch from an exception handler into an enclosing block.

Using SQLCODE and SQLERRM

In an exception handler, you can use the built-in functions SQLCODE and SQLERRM to find out which error occurred and to get the associated error message. For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is *no data found*, in which case SQLCODE returns +100. SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

For user-defined exceptions, SQLCODE returns +1 and SQLERRM returns the message

```
User-Defined Exception
```

unless you used the pragma EXCEPTION_INIT to associate the exception name with an Oracle error number, in which case SQLCODE returns that error number and SQLERRM returns the corresponding error message. The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message

```
ORA-0000: normal, successful completion
```

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass negative error numbers to SQLERRM. In the following example, you pass positive numbers and so get unwanted results:

```
DECLARE
   ...
   err_msg VARCHAR2(100);
BEGIN
   /* Get all Oracle error messages. */
   FOR err_num IN 1..9999 LOOP
      err_msg := SQLERRM(err_num); -- wrong; should be -err_num
      INSERT INTO errors VALUES (err_msg);
   END LOOP;
END;
```

Passing a positive number to SQLERRM always returns the message *user-defined exception* unless you pass +100, in which case SQLERRM returns the message *no data found*. Passing a zero to SQLERRM always returns the message *normal, successful completion*.

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
   err_num NUMBER;
   err_msg VARCHAR2(100);
BEGIN
   ...
EXCEPTION
   ...
   WHEN OTHERS THEN
      err_num := SQLCODE;
      err_msg := SUBSTR(SQLERRM, 1, 100);
      INSERT INTO errors VALUES (err_num, err_msg);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to err_msg. The functions SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

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. 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.

Useful Techniques

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 terminates. 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
      ...
END;
```

Though PL/SQL does not support *continuable* exceptions, you can still handle an exception for a statement, then continue with the next statement. Simply place the statement in its own sub-block with its own exception handlers. If an error occurs in the sub-block, a local handler can catch the exception. When the sub-block terminates, the enclosing block continues to execute at the point where the sub-block ends. Consider the following example:

```
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
      ...
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.

Retrying a Transaction

After an exception is raised, rather than abandon your transaction, you might want to retry it. The technique you use is simple. First, encase the transaction in a sub-block. Then, place the sub-block inside a loop that repeats the transaction.

Before starting the transaction, you mark a savepoint. If the transaction succeeds, you commit, then exit from the loop. If the transaction fails, control transfers to the exception handler, where you roll back to the savepoint undoing any changes, then try to fix the problem.

Consider the example below. When the exception handler completes, the sub-block terminates, control transfers to the LOOP statement in the enclosing block, the sub-block starts executing again, and the transaction is retried. You might want to use a FOR or WHILE loop to limit the number of tries.

```
DECLARE
   name   VARCHAR2(20);
   ans1   VARCHAR2(3);
   ans2   VARCHAR2(3);
   ans3   VARCHAR2(3);
   suffix NUMBER := 1;
BEGIN
   ...
   LOOP  -- could be FOR i IN 1..10 LOOP to allow ten tries
      BEGIN  -- sub-block begins
         SAVEPOINT start_transaction;  -- mark a savepoint
         /* Remove rows from a table of survey results. */
         DELETE FROM results WHERE answer1 = 'NO';
         /* Add a survey respondent's name and answers. */
         INSERT INTO results VALUES (name, ans1, ans2, ans3);
            -- raises DUP_VAL_ON_INDEX if two respondents
            -- have the same name
         COMMIT;
         EXIT;
```

```
      EXCEPTION
         WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK TO start_transaction;  -- undo changes
            suffix := suffix + 1;           -- try to fix problem
            name := name || TO_CHAR(suffix);
      END;  -- sub-block ends
   END LOOP;
END;
```

Using Locator Variables

Exceptions can mask the statement that caused an error, as the following example shows:

```
BEGIN
   SELECT ...
   SELECT ...
   SELECT ...
   ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN ...
      -- Which SELECT statement caused the error?
END;
```

Normally, this is not a problem. But, if the need arises, you can use a *locator variable* to track statement execution, as follows:

```
DECLARE
   stmt INTEGER := 1;  -- designates 1st SELECT statement
BEGIN
   SELECT ...
   stmt := 2;  -- designates 2nd SELECT statement
   SELECT ...
   stmt := 3;  -- designates 3rd SELECT statement
   SELECT ...
   ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO errors VALUES ('Error in statement ' || stmt);
END;
```

7

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 use subprograms, which let you name and encapsulate a sequence of statements. Subprograms aid application development by isolating operations. They are like building blocks, which you can use to construct modular, maintainable applications.

Major Topics

What Are Subprograms?

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called *procedures* and *functions*. Generally, you use a procedure to perform an action and a function to compute a value.

Like unnamed or *anonymous* PL/SQL blocks, subprograms have a declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local and cease to exist when you exit the subprogram. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution.

Consider the following procedure named `debit_account`, which debits a bank account:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
   old_balance REAL;
   new_balance REAL;
   overdrawn   EXCEPTION;
BEGIN
   SELECT bal INTO old_balance FROM accts
      WHERE acct_no = acct_id;
   new_balance := old_balance - amount;
   IF new_balance < 0 THEN
      RAISE overdrawn;
   ELSE
      UPDATE accts SET bal = new_balance
         WHERE acct_no = acct_id;
   END IF;
EXCEPTION
   WHEN overdrawn THEN
      ...
END debit_account;
```

When invoked or *called*, this procedure accepts an account number and a debit amount. It uses the account number to select the account balance from the `accts` database table. Then, it uses the debit amount to compute a new balance. If the new balance is less than zero, an exception is raised; otherwise, the bank account is updated.

Advantages of Subprograms

Subprograms provide *extensibility*; that is, they let you tailor the PL/SQL language to suit your needs. For example, if you need a procedure that creates new departments, you can easily write one, as follows:

```
PROCEDURE create_dept (new_dname VARCHAR2, new_loc VARCHAR2) IS
BEGIN
   INSERT INTO dept
      VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END create_dept;
```

Subprograms also provide *modularity*; that is, they let you break a program down into manageable, well-defined logic modules. This supports top-down design and the stepwise refinement approach to problem solving.

In addition, subprograms promote *reusability* and *maintainability*. Once validated, a subprogram can be used with confidence in any number of applications. If its definition changes, only the subprogram is affected. This simplifies maintenance and enhancement.

Finally, subprograms aid *abstraction*, the mental separation from particulars. To use subprograms, you must know what they do, not how they work. Therefore, you can design applications from the top down without worrying about implementation details. Dummy subprograms (stubs) allow you to defer the definition of procedures and functions until you test and debug the main program.

Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the syntax

```
PROCEDURE name [(parameter[, parameter, ...])] IS
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

where `parameter` stands for the following syntax:

```
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype_name
   [{:= | DEFAULT} expression]
```

You cannot constrain (with NOT NULL for instance) the datatype of a parameter. For example, the following declaration of acct_id is illegal because the datatype CHAR is size-constrained:

```
PROCEDURE reconcile (acct_id CHAR(5)) IS ...  -- illegal
```

However, you can use the following workaround to size-constrain parameter types indirectly:

```
DECLARE
   temp CHAR(5);
   SUBTYPE Char5 IS temp%TYPE;
   PROCEDURE reconcile (acct_id Char5) IS ...
```

A procedure has two parts: the *specification* (*spec* for short) and the *body*. The procedure spec begins with the keyword PROCEDURE and ends with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses.

The procedure body begins with the keyword IS and ends with the keyword END followed by an optional procedure name. The procedure body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE, which introduces declarations in an anonymous PL/SQL block, is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). At least one statement must appear in the executable part of a procedure. The NULL statement meets this requirement. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Consider the procedure raise_salary, which increases the salary of an employee by a given amount:

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
   current_salary REAL;
   salary_missing EXCEPTION;
BEGIN
   SELECT sal INTO current_salary FROM emp
      WHERE empno = emp_id;
   IF current_salary IS NULL THEN
      RAISE salary_missing;
   ELSE
      UPDATE emp SET sal = sal + amount
         WHERE empno = emp_id;
   END IF;
```

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO emp_audit VALUES (emp_id, 'No such number');
   WHEN salary_missing THEN
      INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

When called, this procedure accepts an employee number and a salary increase amount. It uses the employee number to select the current salary from the emp database table. If the employee number is not found or if the current salary is null, an exception is raised. Otherwise, the salary is updated.

A procedure is called as a PL/SQL statement. For example, you might call the procedure raise_salary as follows:

```
DECLARE
   emp_id NUMBER;
   amount REAL;
BEGIN
   ...
   raise_salary(emp_id, amount);
```

Functions

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause. You write functions using the syntax

```
FUNCTION name [(parameter[, parameter, ...])] RETURN datatype IS
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

where parameter stands for the following syntax:

```
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype_name
   [{:= | DEFAULT} expression]
```

You cannot constrain (with NOT NULL for exampe) the datatype of a parameter or function result. However, you can use a workaround to size-constrain them indirectly. See "Functions" on page 7-5.

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 result value. Parameter declarations are optional. Functions that take no parameters are written without parentheses.

The function body begins with the keyword IS and ends with the keyword END followed by an optional function name. The function body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). One or more RETURN statements must appear in the executable part of a function. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END. Consider the function sal_ok, which determines if an employee salary is out of range:

```
FUNCTION sal_ok (salary REAL, title REAL) RETURN BOOLEAN IS
   min_sal REAL;
   max_sal REAL;
BEGIN
   SELECT losal, hisal INTO min_sal, max_sal
      FROM sals
      WHERE job = title;
   RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;
```

When called, this function accepts an employee salary and job title. It uses the job title to select range limits from the sals database table. The function identifier, sal_ok, is set to a Boolean value by the RETURN statement. If the salary is out of range, sal_ok is set to FALSE; otherwise, sal_ok is set to TRUE.

A function is called as part of an expression. For example, the function sal_ok might be called as follows:

```
DECLARE
   new_sal REAL;
   new_title VARCHAR2(15);
BEGIN
   ...
   IF sal_ok(new_sal, new_title) THEN ...
```

The function identifier sal_ok acts like a variable whose value depends on the parameters passed to it.

Controlling Sides Effects

To be callable from SQL statements, a function must obey the following "purity" rules, which are meant to control side effects:

- When called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
- When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.
- When called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot execute DDL statements (such as CREATE) because they are followed by an automatic commit.

If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed).

You can use the pragma (compiler directive) RESTRICT_REFERENCES to check for violations of the rules. The pragma asserts that a function does not read and/or write database tables and/or package variables. For example, the following pragma asserts that packaged function credit_ok writes no database state (WNDS) and reads no package state (RNPS):

```
CREATE PACKAGE loans AS
   ...
   FUNCTION credit_ok RETURN BOOLEAN;
   PRAGMA RESTRICT_REFERENCES (credit_ok, WNDS, RNPS);
END loans;
```

Note: A static INSERT, UPDATE, or DELETE statement always violates WNDS. It also violates RNDS (reads no database state) if it reads any columns. A dynamic INSERT, UPDATE, or DELETE statement always violates WNDS and RNDS.

For more information about the purity rules and pragma RESTRICT_REFERENCES, see *Oracle8i Application Developer's Guide - Fundamentals*.

RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. (Do not confuse the RETURN statement with the RETURN clause, which specifies the datatype of the result value in a function spec.)

A subprogram can contain several RETURN statements, none of which need be the last lexical statement. Executing any of them completes the subprogram immediately. However, to have multiple exit points in a subprogram is a poor programming practice.

In procedures, a RETURN statement cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a RETURN statement *must* contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, 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;
```

A function must contain at least one RETURN statement. Otherwise, PL/SQL raises the predefined exception PROGRAM_ERROR at run time.

Declaring Subprograms

You can declare subprograms in any PL/SQL block, subprogram, or package. But, you must declare subprograms at the end of a declarative section after all other program items. For example, the following procedure declaration is misplaced:

```
DECLARE
   PROCEDURE award_bonus (...) IS  -- misplaced; must come last
   BEGIN
      ...
   END;
   rating NUMBER;
   CURSOR c1 IS SELECT * FROM emp;
```

Forward Declarations

PL/SQL requires that you declare an identifier before using it. Therefore, you must declare a subprogram before calling it. For example, the following declaration of procedure `award_bonus` is illegal because `award_bonus` calls procedure `calc_rating`, which is not yet declared when the call is made:

```
DECLARE
   ...
   PROCEDURE award_bonus ( ... ) IS
   BEGIN
      calc_rating( ... );  -- undeclared identifier
      ...
   END;
   PROCEDURE calc_rating ( ... ) IS
   BEGIN
      ...
   END;
```

In this case, you can solve the problem easily by placing procedure `calc_rating` before procedure `award_bonus`. However, the easy solution does not always work. For example, suppose the procedures are mutually recursive (call each other) or you want to define them in alphabetical order. PL/SQL solves the problem by providing a special subprogram declaration called a *forward declaration*. You can use forward declarations to

- define subprograms in logical or alphabetical order
- define mutually recursive subprograms (see "Recursion" on page 7-37)
- group subprograms in a package

A forward declaration consists of a subprogram spec terminated by a semicolon. In the following example, the forward declaration advises PL/SQL that the body of procedure `calc_rating` can be found later in the block:

```
DECLARE
   PROCEDURE calc_rating ( ... );  -- forward declaration
   ...
   /* Define subprograms in alphabetical order. */
   PROCEDURE award_bonus ( ... ) IS
   BEGIN
      calc_rating( ... );
      ...
   END;
   PROCEDURE calc_rating ( ... ) IS
   BEGIN
      ...
   END;
```

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.

In Packages

Forward declarations also let you group logically related subprograms in a package. The subprogram specs go in the package spec, and the subprogram bodies go in the package body, where they are invisible to applications. Thus, packages allow you to hide implementation details. An example follows:

```
CREATE PACKAGE emp_actions AS  -- package spec
   PROCEDURE hire_employee (emp_id INTGER, name VARCHAR2, ...);
   PROCEDURE fire_employee (emp_id INTEGER);
   PROCEDURE raise_salary (emp_id INTEGER, amount REAL);
   ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS  -- package body
   PROCEDURE hire_employee (emp_id INTGER, name VARCHAR2, ...) IS
   BEGIN
      ...
      INSERT INTO emp VALUES (empno, ename, ...);
   END hire_employee;
```

```
   PROCEDURE fire_employee (emp_id INTEGER) IS
   BEGIN
      DELETE FROM emp
         WHERE empno = emp_id;
   END fire_employee;

   PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
      salary REAL;
   BEGIN
      SELECT sal INTO salary FROM emp
         WHERE empno = emp_id;
      ...
   END raise_salary;
   ...
END emp_actions;
```

You can define subprograms in a package body without declaring their specs in the package spec. However, such subprograms can be called only from inside the package. For more information about packages, see Chapter 8.

Stored Subprograms

Generally, tools (such as Oracle Forms) that incorporate the PL/SQL engine can store subprograms locally for later, strictly local execution. However, to become available for general use, subprograms must be stored in an Oracle database.

To create subprograms and store them permanently in an Oracle database, use the `CREATE PROCEDURE` and `CREATE FUNCTION` statements, which you can execute interactively from SQL*Plus. For example, in SQL*Plus, you might create the procedure `fire_employee`, as follows:

```
SQL> list
  1  CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
  2  BEGIN
  3     DELETE FROM emp WHERE empno = emp_id;
  4* END;
SQL> /

Procedure created.
```

For readability, when creating subprograms, you can use the keyword `AS` instead of `IS` in the spec. For more information about creating and using stored subprograms, see *Oracle8i Application Developer's Guide - Fundamentals.*

Actual versus Formal Parameters

Subprograms pass information using *parameters*. The variables or expressions referenced in the parameter list of a subprogram call are *actual* parameters. For example, the following procedure call lists two actual parameters named emp_num and amount:

```
raise_salary(emp_num, amount);
```

The next procedure call shows that expressions can be used as actual parameters:

```
raise_salary(emp_num, merit + cola);
```

The variables declared in a subprogram spec and referenced in the subprogram body are *formal* parameters. For example, the following procedure declares two formal parameters named emp_id and amount:

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
   current_salary  REAL;
   ...
BEGIN
   SELECT sal INTO current_salary FROM emp WHERE empno = emp_id;
   ...
   UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
END raise_salary;
```

A good programming practice is to use different names for actual and formal parameters.

When you call procedure raise_salary, the actual parameters are evaluated and the result values are assigned to the corresponding formal parameters. If necessary, before assigning the value of an actual parameter to a formal parameter, PL/SQL converts the datatype of the value. For example, the following call to raise_salary is valid:

```
raise_salary(emp_num, '2500');
```

The actual parameter and its corresponding formal parameter must have compatible datatypes. For instance, PL/SQL cannot convert between the DATE and REAL datatypes. Also, the result value must be convertible to the new datatype. The following procedure call raises the predefined exception VALUE_ERROR because PL/SQL cannot convert the second actual parameter to a number:

```
raise_salary(emp_num, '$2500');  -- note the dollar sign
```

For more information, see "Datatype Conversion" on page 2-27.

Positional and Named Notation

When calling a subprogram, you can write the actual parameters using either positional or named notation. That is, you can indicate the association between an actual and formal parameter by position or name. So, given the declarations

```
DECLARE
   acct INTEGER;
   amt  REAL;
   PROCEDURE credit_acct (acct_no INTEGER, amount REAL) IS ...
```

you can call the procedure `credit_acct` in four logically equivalent ways:

```
BEGIN
   credit_acct(acct, amt);                  -- positional notation
   credit_acct(amount => amt, acct_no => acct);  -- named notation
   credit_acct(acct_no => acct, amount => amt);  -- named notation
   credit_acct(acct, amount => amt);             -- mixed notation
```

Positional Notation

The first procedure call uses positional notation. The PL/SQL compiler associates the first actual parameter, `acct`, with the first formal parameter, `acct_no`. And, the compiler associates the second actual parameter, `amt`, with the second formal parameter, `amount`.

Named Notation

The second procedure call uses named notation. An arrow (=>) serves as the association operator, which associates the formal parameter to the left of the arrow with the actual parameter to the right of the arrow.

The third procedure call also uses named notation and shows that you can list the parameter pairs in any order. So, you need not know the order in which the formal parameters are listed.

Mixed Notation

The fourth procedure call shows that you can mix positional and named notation. In this case, the first parameter uses positional notation, and the second parameter uses named notation. Positional notation must precede named notation. The reverse is not allowed. For example, the following procedure call is illegal:

```
credit_acct(acct_no => acct, amt);  -- illegal
```

Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, IN (the default), OUT, and IN OUT, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take zero or more arguments (actual parameters) and return a single value. To have a function return multiple values is a poor programming practice. Also, functions should be free from *side effects*, which change the values of variables not local to the subprogram.

IN Mode

An IN parameter lets you pass values to the subprogram being called. Inside the subprogram, an IN parameter acts like a constant. Therefore, it cannot be assigned a value. For example, the following assignment statement causes a compilation error:

```
PROCEDURE debit_account (acct_id IN INTEGER, amount IN REAL) IS
   minimum_purchase CONSTANT REAL DEFAULT 10.0;
   service_charge   CONSTANT REAL DEFAULT 0.50;
BEGIN
   IF amount < minimum_purchase THEN
      amount := amount + service_charge; -- causes compilation error
   END IF;
   ...
END debit_account;
```

The actual parameter that corresponds to an IN formal parameter can be a constant, literal, initialized variable, or expression. Unlike OUT and IN OUT parameters, IN parameters can be initialized to default values. For more information, see "Parameter Default Values" on page 7-19.

OUT Mode

An OUT parameter lets you return values to the caller of a subprogram. Inside the subprogram, an OUT parameter acts like a variable. That means you can use an OUT formal parameter as if it were a local variable. You can change its value or reference the value in any way, as the following example shows:

```
PROCEDURE calc_bonus (emp_id IN INTEGER, bonus OUT REAL) IS
   hire_date     DATE;
   bonus_missing EXCEPTION;
```

```
BEGIN
   SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
      WHERE empno = emp_id;
   IF bonus IS NULL THEN
      RAISE bonus_missing;
   END IF;
   IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
      bonus := bonus + 500;
   END IF;
   ...
EXCEPTION
   WHEN bonus_missing THEN
      ...
END calc_bonus;
```

The actual parameter that corresponds to an OUT formal parameter must be a variable; it cannot be a constant or an expression. For example, the following procedure call is illegal:

```
calc_bonus(7499, salary + commission);  -- causes compilation error
```

An OUT actual parameter can have a value before the subprogram is called. However, the value is lost when you call the subprogram (unless it exits with an unhandled exception).

Like variables, OUT formal parameters are initialized to NULL. So, the datatype of an OUT formal parameter cannot be a subtype defined as NOT NULL (that includes the built-in subtypes NATURALN and POSITIVEN). Otherwise, when you call the subprogram, PL/SQL raises VALUE_ERROR. An example follows:

```
DECLARE
   SUBTYPE Counter IS INTEGER NOT NULL;
   rows Counter := 0;
   PROCEDURE count_emps (n OUT Counter) IS
   BEGIN
      SELECT COUNT(*) INTO n FROM emp;
   END;
BEGIN
   count_emps(rows);  -- raises VALUE_ERROR
```

Before exiting a subprogram, explicitly assign values to all OUT formal parameters. Otherwise, the corresponding actual parameters will be null. If you exit successfully, PL/SQL assigns values to the actual parameters. However, if you exit with an unhandled exception, PL/SQL does *not* assign values to the actual parameters.

IN OUT Mode

An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, an IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable.

The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or an expression.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. However, if you exit with an unhandled exception, PL/SQL does *not* assign values to the actual parameters.

Summary

Table 7–1 summarizes all you need to know about the parameter modes.

Table 7–1 Parameter Modes

IN	OUT	IN OUT
the default	must be specified	must be specified
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 a 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

Note: The NOCOPY compiler hint allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. For more information, see "NOCOPY Compiler Hint" on page 7-17.

NOCOPY Compiler Hint

Suppose a subprogram declares an IN parameter, an OUT parameter, and an IN OUT parameter. When you call the subprogram, the IN parameter is passed by reference. That is, a pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both parameters reference the same memory location, which holds the value of the actual parameter.

By default, the OUT and IN OUT parameters are passed by value. That is, the value of the IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory. To prevent that, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference.

In the following example, you ask the compiler to pass IN OUT parameter my_staff by reference instead of by value:

```
DECLARE
   TYPE Staff IS VARRAY(200) OF Employee;
   PROCEDURE reorganize (my_staff IN OUT NOCOPY Staff) IS ...
```

Remember, NOCOPY is a hint, not a directive. So, the compiler might pass my_staff by value despite your request. Usually, however, NOCOPY succeeds. So, it can benefit any PL/SQL application that passes around large data structures.

In the example below, 5000 records are loaded into a local nested table, which is passed to two local procedures that do nothing but execute NULL statements. However, a call to one procedure takes 26 seconds because of all the copying. With NOCOPY, a call to the other procedure takes much less than 1 second.

```
SQL> SET SERVEROUTPUT ON
SQL> GET test.sql
  1  DECLARE
  2     TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE;
  3     emp_tab EmpTabTyp := EmpTabTyp(NULL);  -- initialize
  4     t1 CHAR(5);
  5     t2 CHAR(5);
  6     t3 CHAR(5);
  7     PROCEDURE get_time (t OUT NUMBER) IS
  8     BEGIN SELECT TO_CHAR(SYSDATE,'SSSSS') INTO t FROM dual; END;
  9     PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
 10     BEGIN NULL; END;
```

```
 11     PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
 12     BEGIN NULL; END;
 13  BEGIN
 14     SELECT * INTO emp_tab(1) FROM emp WHERE empno = 7788;
 15     emp_tab.EXTEND(4999, 1);  -- copy element 1 into 2..5000
 16     get_time(t1);
 17     do_nothing1(emp_tab);  -- pass IN OUT parameter
 18     get_time(t2);
 19     do_nothing2(emp_tab);  -- pass IN OUT NOCOPY parameter
 20     get_time(t3);
 21     DBMS_OUTPUT.PUT_LINE('Call Duration (secs)');
 22     DBMS_OUTPUT.PUT_LINE('--------------------');
 23     DBMS_OUTPUT.PUT_LINE('Just IN OUT: ' || TO_CHAR(t2 - t1));
 24     DBMS_OUTPUT.PUT_LINE('With NOCOPY: ' || TO_CHAR(t3 - t2));
 25* END;
SQL> /
Call Duration (secs)
--------------------
Just IN OUT: 26
With NOCOPY: 0
```

The Trade-Off for Better Performance

NOCOPY lets you trade well-defined exception semantics for better performance. Its use affects exception handling in the following ways:

- Because NOCOPY is a hint, not a directive, the compiler can pass NOCOPY parameters to a subprogram by value or by reference. So, if the subprogram exits with an unhandled exception, you cannot rely on the values of the NOCOPY actual parameters.
- By default, if a subprogram exits with an unhandled exception, the values assigned to its OUT and IN OUT formal parameters are *not* copied into the corresponding actual parameters, and changes appear to roll back. However, when you specify NOCOPY, assignments to the formal parameters immediately affect the actual parameters as well. So, if the subprogram exits with an unhandled exception, the (possibly unfinished) changes are not "rolled back."
- Currently, RPC protocol lets you pass parameters only by value. So, exception semantics can change silently when you partition applications. For example, if you move a local procedure with NOCOPY parameters to a remote site, those parameters will no longer be passed by reference.

Also, the use of NOCOPY increases the likelihood of parameter aliasing. For more information, see "Parameter Aliasing" on page 7-21.

Restrictions on NOCOPY

In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method (no error is generated):

- The actual parameter is an element of an index-by table. This restriction does not apply to entire index-by tables.
- The actual parameter is constrained (by scale or NOT NULL for example). This restriction does not extend to constrained elements or attributes. Also, it does not apply to size-constrained character strings.
- The actual and formal parameters are records, one or both records were declared using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit datatype conversion.
- The subprogram is involved in an external or remote procedure call.

Parameter Default Values

As the example below shows, you can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

```
PROCEDURE create_dept (
    new_dname CHAR DEFAULT 'TEMP',
    new_loc   CHAR DEFAULT 'TEMP') IS
BEGIN
   INSERT INTO dept
      VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
   ...
END;
```

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the following calls to create_dept:

```
create_dept;
create_dept('MARKETING');
create_dept('MARKETING', 'NEW YORK');
```

The first call passes no actual parameters, so both default values are used. The second call passes one actual parameter, so the default value for new_loc is used. The third call passes two actual parameters, so neither default value is used.

Usually, you can use positional notation to override the default values of formal parameters. However, you cannot skip a formal parameter by leaving out its actual parameter. For example, the following call incorrectly associates the actual parameter 'NEW YORK' with the formal parameter new_dname:

```
create_dept('NEW YORK');  -- incorrect
```

You cannot solve the problem by leaving a placeholder for the actual parameter. For example, the following call is illegal:

```
create_dept(, 'NEW YORK');  -- illegal
```

In such cases, you must use named notation, as follows:

```
create_dept(new_loc => 'NEW YORK');
```

Also, you cannot assign a null to an uninitialized formal parameter by leaving out its actual parameter. For example, given the declaration

```
DECLARE
   FUNCTION gross_pay (
      emp_id   IN NUMBER,
      st_hours IN NUMBER DEFAULT 40,
      ot_hours IN NUMBER) RETURN REAL IS
   BEGIN
      ...
   END;
```

the following function call does not assign a null to ot_hours:

```
IF gross_pay(emp_num) > max_pay THEN ...  -- illegal
```

Instead, you must pass the null explicitly, as in

```
IF gross_pay(emp_num, ot_hour => NULL) > max_pay THEN ...
```

or you can initialize ot_hours to NULL, as follows:

```
ot_hours IN NUMBER DEFAULT NULL;
```

Finally, when creating a stored subprogram, you cannot use host variables (bind variables) in the DEFAULT clause. The following SQL*Plus example causes a *bad bind variable* error because at the time of creation, num is just a placeholder whose value might change:

```
SQL> VARIABLE num NUMBER
SQL> CREATE FUNCTION gross_pay (emp_id IN NUMBER DEFAULT :num, ...
```

Parameter Aliasing

To optimize a subprogram call, the PL/SQL compiler can choose between two methods of parameter passing. With the *by-value* method, the value of an actual parameter is passed to the subprogram. With the *by-reference* method, only a pointer to the value is passed, in which case the actual and formal parameters reference the same item.

The NOCOPY compiler hint increases the possibility of *aliasing* (that is, having two different names refer to the same memory location). This can occur when a global variable appears as an actual parameter in a subprogram call and then is referenced within the subprogram. The result is indeterminate because it depends on the method of parameter passing chosen by the compiler.

In the example below, procedure add_entry refers to varray lexicon in two different ways: as a parameter and as a global variable. So, when add_entry is called, the identifiers word_list and lexicon name the same varray.

```
DECLARE
   TYPE Definition IS RECORD (
      word    VARCHAR2(20),
      meaning VARCHAR2(200));
   TYPE Dictionary IS VARRAY(2000) OF Definition;
   lexicon Dictionary := Dictionary();
   PROCEDURE add_entry (word_list IN OUT NOCOPY Dictionary) IS
   BEGIN
     word_list(1).word := 'aardvark';
     lexicon(1).word := 'aardwolf';
   END;
BEGIN
   lexicon.EXTEND;
   add_entry(lexicon);
   DBMS_OUTPUT.PUT_LINE(lexicon(1).word);
      -- prints 'aardvark' if parameter was passed by value
      -- prints 'aardwolf' if parameter was passed by reference
END;
```

The result depends on the method of parameter passing chosen by the compiler. If the compiler chooses the by-value method, word_list and lexicon are separate copies of the same varray. So, changing one does not affect the other. But, if the compiler chooses the by-reference method, word_list and lexicon are just different names for the same varray. (Hence, the term "aliasing.") So, changing the value of lexicon(1) also changes the value of word_list(1).

Aliasing can also occur when the same actual parameter appears more than once in a subprogram call. In the example below, n2 is an IN OUT parameter, so the value of the actual parameter is not updated until the procedure exits. That is why the first PUT_LINE prints 10 (the initial value of n) and the third PUT_LINE prints 20. However, n3 is a NOCOPY parameter, so the value of the actual parameter is updated immediately. That is why the second PUT_LINE prints 30.

```
DECLARE
   n NUMBER := 10;
   PROCEDURE do_something (
      n1 IN NUMBER,
      n2 IN OUT NUMBER,
      n3 IN OUT NOCOPY NUMBER) IS
   BEGIN
      n2 := 20;
      DBMS_OUTPUT.PUT_LINE(n1);  -- prints 10
      n3 := 30;
      DBMS_OUTPUT.PUT_LINE(n1);  -- prints 30
   END;
BEGIN
   do_something(n, n, n);
   DBMS_OUTPUT.PUT_LINE(n);  -- prints 20
END;
```

Because they are pointers, cursor variables also increase the possibility of aliasing. Consider the example below. After the assignment, emp_cv2 is an alias of emp_cv1 because both point to the same query work area. So, both can alter its state. That is why the first fetch from emp_cv2 fetches the third row (not the first) and why the second fetch from emp_cv2 fails after you close emp_cv1.

```
PROCEDURE get_emp_data (
   emp_cv1 IN OUT EmpCurTyp,
   emp_cv2 IN OUT EmpCurTyp) IS
   emp_rec emp%ROWTYPE;
BEGIN
   OPEN emp_cv1 FOR SELECT * FROM emp;
   emp_cv2 := emp_cv1;
```

```
   FETCH emp_cv1 INTO emp_rec;  -- fetches first row
   FETCH emp_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;
```

Overloading

PL/SQL lets you *overload* subprogram names. That is, you can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

Suppose you want to initialize the first *n* rows in two index-by tables that were declared as follows:

```
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;
```

You might write the following procedure to initialize the index-by table named `hiredate_tab`:

```
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
   END LOOP;
END initialize;
```

And, you might write the next procedure to initialize the index-by table named `sal_tab`:

```
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
      tab(i) := 0.0;
   END LOOP;
END initialize;
```

Because the processing in these two procedures is the same, it is logical to give them the same name.

You can place the two overloaded `initialize` procedures in the same block, subprogram, or package. PL/SQL determines which of the two procedures is being called by checking their formal parameters.

Consider the example below. If you call `initialize` with a `DateTabTyp` parameter, PL/SQL uses the first version of `initialize`. But, if you call `initialize` with a `RealTabTyp` parameter, PL/SQL uses the second version.

```
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;
BEGIN
   indx := 50;
   initialize(hiredate_tab, indx);  -- calls first version
   initialize(comm_tab, indx);      -- calls second version
```

Restrictions

Only local or packaged subprograms can be overloaded. Therefore, you cannot overload stand-alone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode. For example, you cannot overload the following two procedures:

```
PROCEDURE reconcile (acct_no IN INTEGER) IS BEGIN ... END;
PROCEDURE reconcile (acct_no OUT INTEGER) IS BEGIN ... END;
```

Furthermore, you cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family. For instance, you cannot overload the following procedures because the datatypes `INTEGER` and `REAL` are in the same family:

```
PROCEDURE charge_back (amount INTEGER) IS BEGIN ... END;
PROCEDURE charge_back (amount REAL) IS BEGIN ... END;
```

Likewise, you cannot overload two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family. For example, you cannot overload the following procedures because the base types `CHAR` and `LONG` are in the same family:

```
DECLARE
   SUBTYPE Delimiter IS CHAR;
   SUBTYPE Text IS LONG;
```

```
   PROCEDURE scan (x Delimiter) IS BEGIN ... END;
   PROCEDURE scan (x Text) IS BEGIN ... END;
```

Finally, you cannot overload two functions that differ only in return type (the datatype of the result value) even if the types are in different families. For example, you cannot overload the following functions:

```
FUNCTION acct_ok (acct_id INTEGER) RETURN BOOLEAN IS BEGIN ... END;
FUNCTION acct_ok (acct_id INTEGER) RETURN INTEGER IS BEGIN ... END;
```

How Calls Are Resolved

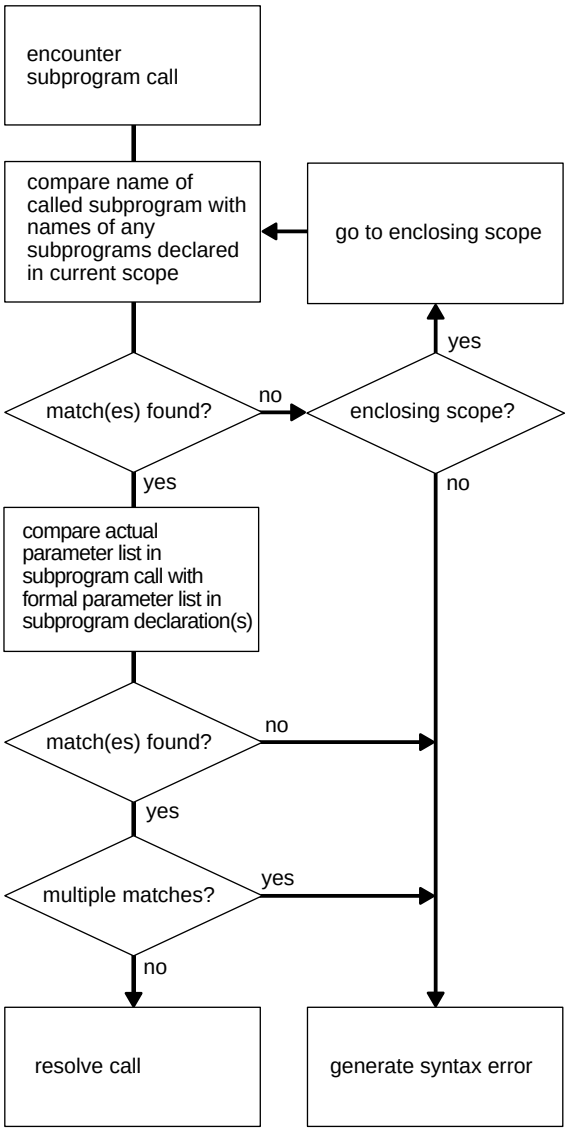
Figure 7–1 shows how the PL/SQL compiler resolves subprogram calls. When the compiler encounters a procedure or function call, it tries to find a declaration that matches the call. The compiler searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the subprogram name matches the name of the called subprogram.

To resolve a call among possibly like-named subprograms at the same level of scope, the compiler must find an *exact* match between the actual and formal parameters. That is, they must match in number, order, and datatype (unless some formal parameters were assigned default values). If no match is found or if multiple matches are found, the compiler generates a syntax error.

In the following example, you call the enclosing procedure `swap` from within the function `valid`. However, the compiler generates an error because neither declaration of `swap` within the current scope matches the procedure call:

```
PROCEDURE swap (d1 DATE, d2 DATE) IS
   date1 DATE;
   date2 DATE;
   FUNCTION valid (d DATE) RETURN BOOLEAN IS
      PROCEDURE swap (n1 INTEGER, n2 INTEGER) IS BEGIN ... END;
      PROCEDURE swap (n1 REAL, n2 REAL) IS BEGIN ... END;
   BEGIN
      ...
      swap(date1, date2);
   END valid;
BEGIN
   ...
END;
```

Figure 7–1 How the PL/SQL Compiler Resolves Calls



Avoiding Call Resolution Errors

PL/SQL declares built-in functions globally in package STANDARD. Redeclaring them locally is error prone because your local declaration overrides the global declaration. Consider the following example, in which you declare a function named `sign`, then within the scope of that declaration, try to call the built-in function `SIGN`:

```
DECLARE
   x NUMBER;
   ...
BEGIN
   DECLARE
      FUNCTION sign (n NUMBER) RETURN NUMBER IS
      BEGIN
         IF n < 0 THEN
            RETURN -1;
         ELSE
            RETURN 1;
         END IF;
      END;
   BEGIN
      ...
      x := SIGN(0);  -- assigns 1 to x
   END;
   ...
   x := SIGN(0);  -- assigns 0 to x
END;
```

Inside the sub-block, PL/SQL uses your function definition, *not* the built-in definition. To call the built-in function from inside the sub-block, you must use dot notation, as follows:

```
x := STANDARD.SIGN(0);  -- assigns 0 to x
```

Calling External Routines

PL/SQL is a powerful development tool; you can use it for almost any purpose. However, it is specialized for SQL transaction processing. So, some tasks are more quickly done in a lower-level language such as C, which is very efficient at machine-precision calculations. Other tasks are more easily done in a fully object-oriented, standardized language such as Java.

To support such special-purpose processing, you can use PL/SQL call specs to invoke *external routines* (that is, routines written in other languages). This makes the strengths and capabilities of those languages available to you. No longer are you restricted to one language with its inherent limitations.

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)';
```

Later, you might call procedure `raise_salary` from an anonymous PL/SQL block, as follows:

```
DECLARE
   emp_id  NUMBER;
   percent NUMBER;
BEGIN
   -- get values for emp_id and percent
   raise_salary(emp_id, percent);  -- call external routine
```

For more information about Java stored procedures, see *Oracle8i Java Stored Procedures Developer's Guide*.

Typically, external C routines are used to interface with embedded systems, solve engineering problems, analyze data, or control real-time devices and processes. Moreover, external C routines enable you to extend the functionality of the database server, and to move computation-bound programs from client to server, where they will execute faster.

For more information about external C routines, see *Oracle8i Application Developer's Guide - Fundamentals*.

Invoker Rights versus Definer Rights

By default, stored procedures and SQL methods execute with the privileges of their definer, not their invoker. Such *definer-rights* routines are bound to the schema in which they reside. For example, assume that duplicates of table `dept` reside in schema `scott` and schema `blake`, and that the following stand-alone procedure resides in schema `scott`:

```
CREATE PROCEDURE create_dept (new_dname CHAR, new_loc CHAR) AS
BEGIN
   INSERT INTO dept
      VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END;
```

Also assume that user `scott` has granted the `EXECUTE` privilege on this procedure to user `blake`. When user `blake` calls the procedure, the `INSERT` statement executes with the privileges of user `scott`. Also, the unqualified reference to table `dept` is resolved in schema `scott`. So, the procedure updates the `dept` table in schema `scott`, not schema `blake`.

How can routines in one schema manipulate objects in another schema? One way is to fully qualify references to the objects, as in

```
INSERT INTO blake.dept ...
```

However, that hampers portability. Another way is to copy the routines into the other schema. However, that hampers maintenance. A better way is to use the `AUTHID` clause, which enables stored procedures and SQL methods to execute with the privileges of their invoker (current user). Such *invoker-rights* routines are not bound to a particular schema. They can be run by a variety of users, and their definer need not know who those users will be.

For example, the following version of procedure `create_dept` executes with the privileges of its invoker:

```
CREATE PROCEDURE create_dept (new_dname CHAR, new_loc CHAR)
   AUTHID CURRENT_USER AS
BEGIN
   INSERT INTO dept
      VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END;
```

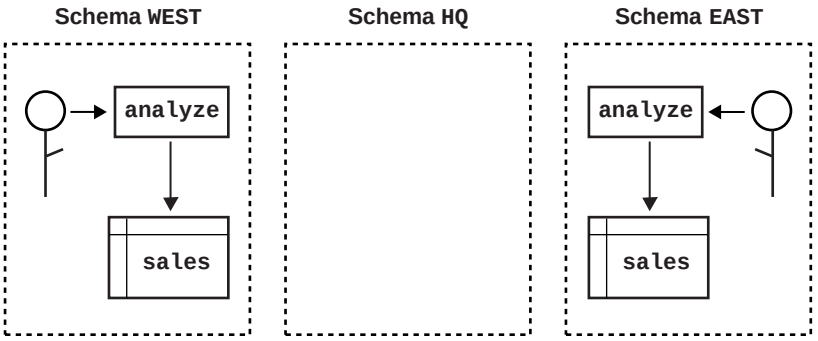
Also, the unqualified reference to table `dept` is resolved in the schema of the invoker, not the definer.

Advantages of Invoker Rights

Invoker-rights routines let you centralize data retrieval. They are especially useful in applications that store data in different schemas. In such cases, multiple users can manage their own data using a single code base.

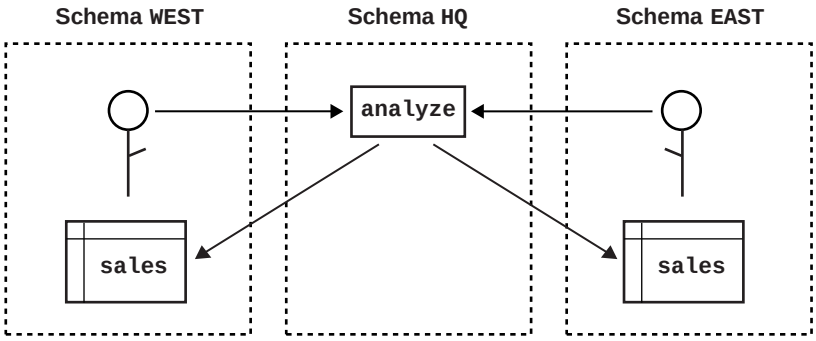
Consider a company that uses a definer-rights procedure to analyze sales. To provide local sales statistics, procedure `analyze` must access `sales` tables that reside at each regional site. So, as Figure 7–2 shows, the procedure must also reside at each regional site. This causes a maintenance problem.

Figure 7–2 Definer Rights Problem



To solve the problem, the company installs an invoker-rights version of procedure `analyze` at headquarters. Now, as Figure 7–3 shows, all regional sites can use the same procedure to query their own `sales` tables.

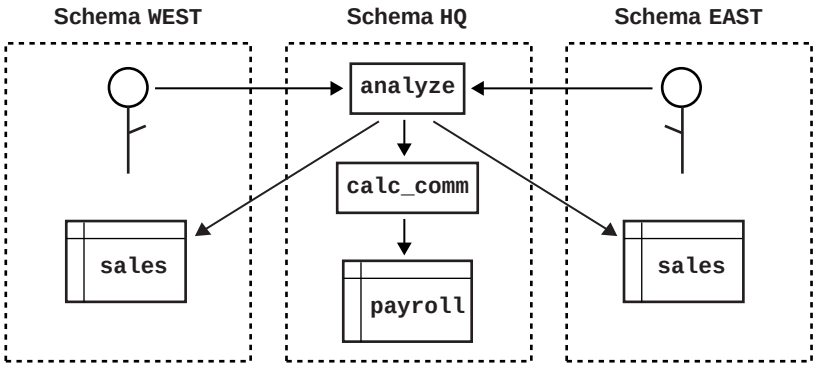
Figure 7–3 Invoker Rights Solution



Invoker-rights routines also let you restrict access to sensitive data. Suppose headquarters would like procedure `analyze` to calculate sales commissions and update a central `payroll` table.

That presents a problem because invokers of `analyze` should not have direct access to the `payroll` table, which stores employee salaries and other sensitive data. As Figure 7–4 shows, the solution is to have procedure `analyze` call definer-rights procedure `calc_comm`, which in turn updates the `payroll` table.

Figure 7–4 Indirect Access



Using the AUTHID Clause

To implement invoker rights, use the AUTHID clause, which specifies whether a routine executes with the privileges of its definer or its invoker. It also specifies whether *external references* (that is, references to objects outside the routine) are resolved in the schema of the definer or the invoker.

The AUTHID clause is allowed only in the header of a stand-alone subprogram, a package spec, or an object type spec. The header syntax follows:

```
-- stand-alone function
CREATE [OR REPLACE] FUNCTION [schema_name.]function_name
[(parameter_list)] RETURN datatype
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- stand-alone procedure
CREATE [OR REPLACE] PROCEDURE [schema_name.]procedure_name
[(parameter_list)]
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- package spec
CREATE [OR REPLACE] PACKAGE [schema_name.]package_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}

-- object type spec
CREATE [OR REPLACE] TYPE [schema_name.]object_type_name
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT
```

where DEFINER is the default option. In a package or object type, the AUTHID clause applies to all routines.

How External References Are Resolved

If you specify AUTHID CURRENT_USER, the privileges of the invoker are checked at run time, and external references are resolved in the schema of the invoker. 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 definer are checked at compile time, and external references are resolved in the schema of the definer. For example, the assignment statement below refers to a packaged function. This external reference is resolved in the schema of the definer of procedure `reconcile`.

```
CREATE PROCEDURE reconcile (acc_id IN INTEGER)
   AUTHID CURRENT_USER AS
   bal NUMBER;
BEGIN
   bal := bank_stuff.balance(acct_id);
   ...
END;
```

Avoiding Name Resolution Errors

External references in an invoker-rights routine are resolved in the schema of the invoker at run time. However, the PL/SQL compiler must resolve all references at compile time. So, the definer must create template objects in his or her schema beforehand. At run time, the template objects and the actual objects must match. Otherwise, you get an error or unexpected results.

For example, suppose user `scott` creates the following database table and stand-alone procedure:

```
CREATE TABLE emp (
   empno NUMBER(4),
   ename VARCHAR2(15));
/
CREATE PROCEDURE evaluate (my_empno NUMBER)
   AUTHID CURRENT_USER AS
   my_ename VARCHAR2(15);
BEGIN
   SELECT ename INTO my_ename FROM emp WHERE empno = my_empno;
   ...
END;
/
```

Now, suppose user `blake` creates a similar database table, then calls procedure `evaluate`, as follows:

```
CREATE TABLE emp (
   empno    NUMBER(4),
   ename    VARCHAR2(15),
   my_empno NUMBER(4));  -- note extra column
/
```

```
DECLARE
   ...
BEGIN
   ...
   scott.evaluate(7788);
END;
/
```

The procedure call executes without error but ignores column my_empno in the table created by user blake. That is because the actual table in the schema of the invoker does not match the template table used at compile time.

Overriding Default Name Resolution

Occasionally, you might want to override the default invoker-rights behavior. Suppose user scott defines the stand-alone procedure below. Notice that the SELECT statement calls an external function. Normally, this external reference would be resolved in the schema of the invoker.

```
CREATE PROCEDURE calc_bonus (emp_id INTEGER)
   AUTHID CURRENT_USER AS
   mid_sal REAL;
BEGIN
   SELECT median(sal) INTO mid_sal FROM emp;
   ...
END;
```

To have the reference resolved in the schema of the definer, create a public synonym for the function using the CREATE SYNONYM statement, as follows:

```
CREATE PUBLIC SYNONYM median FOR scott.median;
```

This works unless the invoker has defined a function or private synonym named median.

Alternatively, you can fully qualify the reference, as follows:

```
BEGIN
   SELECT scott.median(sal) INTO mid_sal FROM emp;
   ...
END;
```

This works unless the invoker has defined a package named scott that contains a function named median.

Granting the EXECUTE Privilege

To call an invoker-rights routine directly, users must have the EXECUTE privilege on that routine. By granting the privilege, you allow a user to

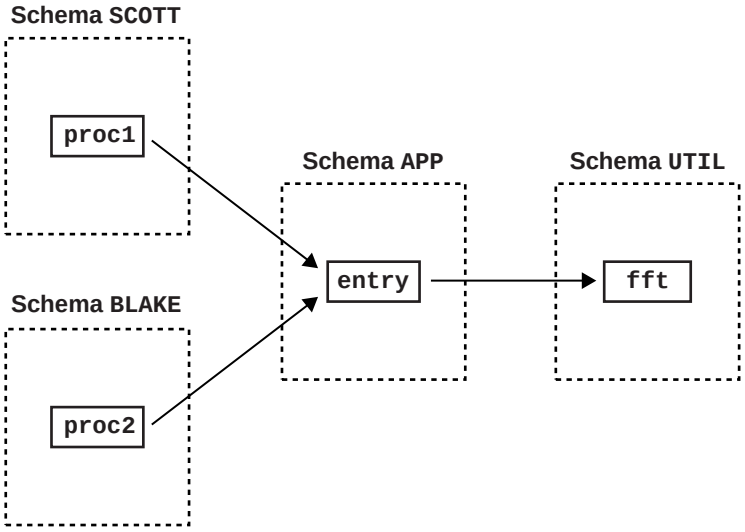
- call the routine directly
- compile functions and procedures that call the routine

Suppose user `util` grants the EXECUTE privilege on invoker-rights routine `fft` to user `app`, as follows:

```
GRANT EXECUTE ON util.fft TO app;
```

Now, user `app` can compile functions and procedures that call routine `fft`. At run time, no privilege checks on the calls are done. So, as Figure 7–5 shows, user `util` need not grant the EXECUTE privilege to every user who might call `fft` indirectly.

Figure 7–5 Indirect Calls to an Invoker-Rights Routine



Notice that routine `util.fft` is called directly only from definer-rights routine `app.entry`. So, user `util` must grant the EXECUTE privilege only to user `app`. When `util.fft` is executed, its invoker might be `app`, `scott`, or `blake`.

Using Views and Database Triggers

For invoker-rights routines executed within a view expression, the view owner, not the view user, is considered to be the invoker. For example, suppose user `scott` creates a view, as follows:

```
CREATE VIEW payroll AS SELECT layout(3) FROM dual;
```

The function `layout` always executes with the privileges of user `scott`, who is the view owner.

This rule also applies to database triggers.

Using Database Links

External references to database links are resolved in the usual way. However, the link determines which user is connected to the remote database.

With named links, the username specified in the link is used. Suppose a routine owned by user `blake` references the database link below. No matter who calls the routine, it connects to the Chicago database as user `scott`.

```
CREATE DATABASE LINK chicago CONNECT TO scott
   IDENTIFIED BY tiger USING connect_string;
```

With anonymous links, the session username is used. Suppose a routine owned by user `blake` references the database link below. If user `scott` logs on, and then calls the routine, it connects to the Atlanta database as user `scott`.

```
CREATE DATABASE LINK atlanta USING connect_string;
```

With privileged links, the username of the invoker is used. Suppose an invoker-rights routine owned by user `blake` references the database link below. If global user `scott` calls the routine, it connects to the Dallas database as user `scott`, who is the current user.

```
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER
   USING connect_string;
```

If it were a definer-rights routine, the current user would be `blake`. So, the routine would connect to the Dallas database as global user `blake`.

Invoking Instance Methods

An invoker-rights instance method executes with the privileges of the invoker, not the creator of the instance. Suppose that `Person` is an invoker-rights object type, and that user `scott` creates `p1`, an object of type `Person`. If user `blake` calls instance method `change_job` to operate on object `p1`, the invoker 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;
/
```

Recursion

Recursion is a powerful technique for simplifying the design of algorithms. Basically, *recursion* means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, ...), which was first used to model the growth of a rabbit colony, is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined in terms of simpler versions of itself. Consider the definition of *n* factorial (*n!*), the product of all integers from 1 to *n*:

```
n! = n * (n - 1)!
```

Recursive Subprograms

A recursive subprogram is one that calls itself. Think of a recursive call as a call to some other subprogram that does the same task as your subprogram. Each recursive call creates a new instance of any items declared in the subprogram, including parameters, variables, cursors, and exceptions. Likewise, new instances of SQL statements are created at each level in the recursive descent.

Be careful where you place a recursive call. If you place it inside a cursor FOR loop or between OPEN and CLOSE statements, another cursor is opened at each call. As a result, your program might exceed the limit set by the Oracle initialization parameter OPEN_CURSORS.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a *terminating condition*. Otherwise, the recursion would (theoretically) go on forever. In practice, if a recursive subprogram strays into infinite regress, PL/SQL eventually runs out of memory and raises the predefined exception STORAGE_ERROR.

Example 1

To solve some programming problems, you must repeat a sequence of statements until a condition is met. You can use iteration or recursion to solve such problems. Use recursion when the problem can be broken down into simpler versions of itself. For example, you can evaluate 3! as follows:

```
0! = 1  -- by definition
1! = 1 * 0! = 1
2! = 2 * 1! = 2
3! = 3 * 2! = 6
```

To implement this algorithm, you might write the following recursive function, which returns the factorial of a positive integer:

```
FUNCTION fac (n POSITIVE) RETURN INTEGER IS  -- returns n!
BEGIN
   IF n = 1 THEN  -- terminating condition
      RETURN 1;
   ELSE
      RETURN n * fac(n - 1);  -- recursive call
   END IF;
END fac;
```

At each recursive call, n is decremented. Eventually, n becomes 1 and the recursion stops.

Example 2

Consider the procedure below, which finds the staff of a given manager. The procedure declares two formal parameters, `mgr_no` and `tier`, which represent the manager's employee number and a tier in his or her departmental organization. Staff members reporting directly to the manager occupy the first tier.

```
PROCEDURE find_staff (mgr_no NUMBER, tier NUMBER := 1) IS
   boss_name  CHAR(10);
   CURSOR c1 (boss_no NUMBER) IS
      SELECT empno, ename FROM emp WHERE mgr = boss_no;
BEGIN
   /* Get manager's name. */
   SELECT ename INTO boss_name FROM emp WHERE empno = mgr_no;
   IF tier = 1 THEN
      INSERT INTO staff  -- single-column output table
         VALUES (boss_name || ' manages the staff');
   END IF;
   /* Find staff members who report directly to manager. */
   FOR ee IN c1 (mgr_no) LOOP
      INSERT INTO staff
         VALUES (boss_name || ' manages ' || ee.ename
            || ' on tier ' || TO_CHAR(tier));
      /* Drop to next tier in organization. */
      find_staff(ee.empno, tier + 1);  -- recursive call
   END LOOP;
   COMMIT;
END;
```

When called, the procedure accepts a value for `mgr_no` but uses the default value of `tier`. For example, you might call the procedure as follows:

```
find_staff(7839);
```

The procedure passes `mgr_no` to a cursor in a cursor `FOR` loop, which finds staff members at successively lower tiers in the organization. At each recursive call, a new instance of the `FOR` loop is created and another cursor is opened, but prior cursors stay positioned on the next row in their result sets.

When a fetch fails to return a row, the cursor is closed automatically and the `FOR` loop is exited. Since the recursive call is inside the `FOR` loop, the recursion stops. Unlike the initial call, each recursive call passes a second actual parameter (the next tier) to the procedure.

The last example illustrates recursion, not the efficient use of set-oriented SQL statements. You might want to compare the performance of the recursive procedure to that of the following SQL statement, which does the same task:

```
INSERT INTO staff
   SELECT PRIOR ename || ' manages ' || ename
      || ' on tier ' || TO_CHAR(LEVEL - 1)
   FROM emp
   START WITH empno = 7839
   CONNECT BY PRIOR empno = mgr;
```

The SQL statement is appreciably faster. However, the procedure is more flexible. For example, a multi-table query cannot contain the CONNECT BY clause. So, unlike the procedure, the SQL statement cannot be modified to do joins. (A *join* combines rows from two or more database tables.) In addition, a procedure can process data in ways that a single SQL statement cannot.

Mutual Recursion

Subprograms are *mutually recursive* if they directly or indirectly call each other. In the example below, the Boolean functions odd and even, which determine whether a number is odd or even, call each other directly. The forward declaration of odd is necessary because even calls odd, which is not yet declared when the call is made. (See "Forward Declarations" on page 7-9.)

```
FUNCTION odd (n NATURAL) RETURN BOOLEAN;  -- forward declaration

FUNCTION even (n NATURAL) RETURN BOOLEAN IS
BEGIN
   IF n = 0 THEN
      RETURN TRUE;
   ELSE
      RETURN odd(n - 1);  -- mutually recursive call
   END IF;
END even;

FUNCTION odd (n NATURAL) RETURN BOOLEAN IS
BEGIN
   IF n = 0 THEN
      RETURN FALSE;
   ELSE
      RETURN even(n - 1);  -- mutually recursive call
   END IF;
END odd;
```

When a positive integer n is passed to odd or even, the functions call each other by turns. At each call, n is decremented. Ultimately, n becomes zero and the final call returns TRUE or FALSE. For instance, passing the number 4 to odd results in this sequence of calls:

```
odd(4)
even(3)
odd(2)
even(1)
odd(0)  -- returns FALSE
```

On the other hand, passing the number 4 to even results in this sequence of calls:

```
even(4)
odd(3)
even(2)
odd(1)
even(0)  -- returns TRUE
```

Recursion versus Iteration

Unlike iteration, recursion is not essential to PL/SQL programming. Any problem that can be solved using recursion can be solved using iteration. Also, the iterative version of a subprogram is usually easier to design than the recursive version. However, the recursive version is usually simpler, smaller, and therefore easier to debug. Compare the following functions, which compute the *n*th Fibonacci number:

```
-- recursive version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
BEGIN
   IF (n = 1) OR (n = 2) THEN
      RETURN 1;
   ELSE
      RETURN fib(n - 1) + fib(n - 2);
   END IF;
END fib;

-- iterative version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
   pos1 INTEGER := 1;
   pos2 INTEGER := 0;
   cum  INTEGER;
```

```
BEGIN
   IF (n = 1) OR (n = 2) THEN
      RETURN 1;
   ELSE
      cum := pos1 + pos2;
      FOR i IN 3..n LOOP
         pos2 := pos1;
         pos1 := cum;
         cum := pos1 + pos2;
      END LOOP;
      RETURN cum;
   END IF;
END fib;
```

The recursive version of `fib` is more elegant than the iterative version. However, the iterative version is more efficient; it runs faster and uses less storage. That is because each recursive call requires additional time and memory. As the number of recursive calls gets larger, so does the difference in efficiency. Still, if you expect the number of recursive calls to be small, you might choose the recursive version for its readability.

8

Packages

Goods which are not shared are not goods. —Fernando de Rojas

This chapter shows you how to bundle related PL/SQL programming resource into a package. The resources might include a collection of procedures or a pool of type definitions and variable declarations. For example, a Human Resources package might contain hiring and firing procedures. Once written, your general-purpose package is compiled, then stored in an Oracle database, where its contents can be shared by many applications.

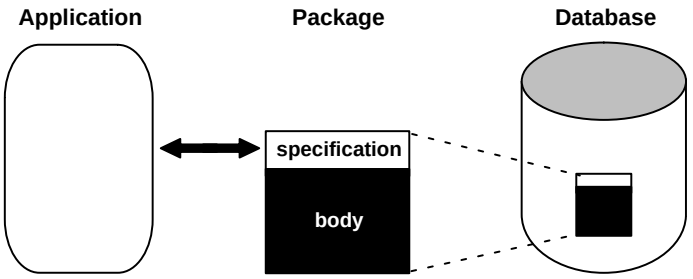
Major Topics

What Is a Package?

A *package* is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The *specification* (*spec* for short) is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The *body* fully defines cursors and subprograms, and so implements the spec.

As Figure 8–1 shows, you can think of the spec as an operational interface and of the body as a "black box." You can debug, enhance, or replace a package body without changing the interface (package spec) to the package.

Figure 8–1 Package Interface



To create packages, use the CREATE PACKAGE statement, which you can execute interactively from SQL*Plus. Here is the syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_spec [cursor_spec] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
END [package_name];

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_body [cursor_body] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
[BEGIN
  sequence_of_statements]
END [package_name];]
```

The spec holds *public* declarations, which are visible to your application. The body holds implementation details and *private* declarations, which are hidden from your application. Following the declarative part of the package body is the optional initialization part, which typically holds statements that initialize package variables.

The AUTHID clause determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights versus Definer Rights" on page 7-29.

A *call spec* lets you publish a Java method or external C function in the Oracle data dictionary. The call spec publishes the routine by mapping its name, parameter types, and return type to their SQL counterparts. To learn how to write Java call specs, see *Oracle8i Java Stored Procedures Developer's Guide*. To learn how to write C call specs, see *Oracle8i Application Developer's Guide - Fundamentals*.

In the example below, you package a record type, a cursor, and two employment procedures. Notice that the procedure hire_employee uses the database sequence empno_seq and the function SYSDATE to insert a new employee number and hire date, respectively.

```
CREATE OR REPLACE PACKAGE emp_actions AS  -- spec
   TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
   CURSOR desc_salary RETURN EmpRecTyp;
   PROCEDURE hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    NUMBER,
      sal    NUMBER,
      comm   NUMBER,
      deptno NUMBER);
   PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- body
   CURSOR desc_salary RETURN EmpRecTyp IS
      SELECT empno, sal FROM emp ORDER BY sal DESC;
   PROCEDURE hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    NUMBER,
      sal    NUMBER,
      comm   NUMBER,
      deptno NUMBER) IS
```

```
   BEGIN
      INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,
         mgr, SYSDATE, sal, comm, deptno);
   END hire_employee;
   PROCEDURE fire_employee (emp_id NUMBER) IS
   BEGIN
      DELETE FROM emp WHERE empno = emp_id;
   END fire_employee;
END emp_actions;
```

Only the declarations in the package spec are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. So, you can change the body (implementation) without having to recompile calling programs.

Advantages of Packages

Packages offer several advantages: modularity, easier application design, information hiding, added functionality, and better performance.

Modularity

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.

Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the definition of the private subprogram so that only the package (not your application) is affected if the definition changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

Added Functionality

Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.

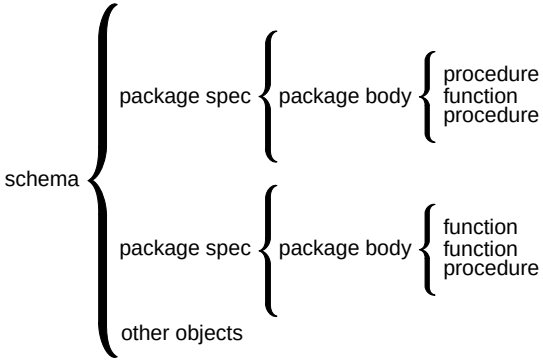
Better Performance

When you call a packaged subprogram for the first time, the whole package is loaded into memory. So, later calls to related subprograms in the package require no disk I/O. Also, packages stop cascading dependencies and thereby avoid unnecessary recompiling. For example, if you change the definition of a packaged function, Oracle need not recompile the calling subprograms because they do not depend on the package body.

The Package Spec

The package spec contains public declarations. The scope of these declarations is local to your database schema and global to the package. So, the declared items are accessible from your application and from anywhere in the package. Figure 8–2 illustrates the scoping.

Figure 8–2 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).

Only subprograms and cursors have an underlying implementation or *definition*. So, if a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. Consider the following bodiless package:

```
-- a bodiless package
CREATE PACKAGE trans_data AS
   TYPE TimeRec IS RECORD (
      minutes SMALLINT,
      hours   SMALLINT);
   TYPE TransRec IS RECORD (
      category VARCHAR2,
      account  INTEGER,
      amount   REAL,
      time_of  TimeRec);
   minimum_balance    CONSTANT REAL := 10.00;
   number_processed   INTEGER;
   insufficient_funds EXCEPTION;
END trans_data;
```

The package `trans_data` needs no body because types, constants, variables, and exceptions do not have an underlying implementation. Such packages let you define global variables—usable by subprograms and database triggers—that persist throughout a session.

Referencing Package Contents

To reference the types, items, subprograms, and call specs declared within a package spec, use dot notation, as follows:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
package_name.call_spec_name
```

You can reference package contents from database triggers, stored subprograms, 3GL application programs, and various Oracle tools. For example, you might call the packaged procedure `hire_employee` from SQL*Plus, as follows:

```
SQL> CALL emp.actions.hire_employee('TATE', 'CLERK', ...);
```

In the example below, you call the same procedure from an anonymous PL/SQL block embedded in a Pro*C program. The actual parameters `emp_name` and `job_title` are host variables (that is, variables declared in a host environment).

```
EXEC SQL EXECUTE
   BEGIN
      emp_actions.hire_employee(:emp_name, :job_title, ...);
```

Restrictions

You cannot reference remote packaged variables directly or indirectly. For example, you cannot call the following procedure remotely because it references a packaged variable in a parameter initialization clause:

```
CREATE PACKAGE random AS
   seed NUMBER;
   PROCEDURE initialize (starter IN NUMBER := seed, ...);
```

Also, inside a package, you cannot reference host variables.

The Package Body

The package body implements the package spec. That is, the package body contains the definition of every cursor and subprogram declared in the package spec. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specs also appear in the package spec.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. So, except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception, as the following example shows:

```
CREATE PACKAGE emp_actions AS
   ...
   PROCEDURE calc_bonus (date_hired emp.hiredate%TYPE, ...);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
   ...
   PROCEDURE calc_bunus (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

Consider the package below named `emp_actions`. The package spec declares the following types, items, and subprograms:

- types `EmpRecTyp` and `DeptRecTyp`
- cursor `desc_salary`
- exception `salary_missing`
- functions `hire_employee`, `nth_highest_salary`, and `rank`
- procedures `fire_employee` and `raise_salary`

After writing the package, you can develop applications that reference its types, call its subprograms, use its cursor, and raise its exception. When you create the package, it is stored in an Oracle database for general use.

```
CREATE PACKAGE emp_actions AS
   /* Declare externally visible types, cursor, exception. */
   TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
   TYPE DeptRecTyp IS RECORD (dept_id INTEGER, location VARCHAR2);
   CURSOR desc_salary RETURN EmpRecTyp;
   salary_missing EXCEPTION;
```

```
   /* Declare externally callable subprograms. */
   FUNCTION hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    NUMBER,
      sal    NUMBER,
      comm   NUMBER,
      deptno NUMBER) RETURN INTEGER;
   PROCEDURE fire_employee (emp_id INTEGER);
   PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER);
   FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
   number_hired  INTEGER;  -- 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    NUMBER,
      sal    NUMBER,
      comm   NUMBER,
      deptno NUMBER) RETURN INTEGER IS
      new_empno  INTEGER;
   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 INTEGER) IS
   BEGIN
      DELETE FROM emp WHERE empno = emp_id;
   END fire_employee;
```

```
   PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER) IS
      current_salary NUMBER;
   BEGIN
      SELECT sal INTO current_salary FROM emp
         WHERE empno = emp_id;
      IF current_salary IS NULL THEN
         RAISE salary_missing;
      ELSE
         UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
      END IF;
   END raise_salary;

   FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
      emp_rec  EmpRecTyp;
   BEGIN
      OPEN desc_salary;
      FOR i IN 1..n LOOP
         FETCH desc_salary INTO emp_rec;
      END LOOP;
      CLOSE desc_salary;
      RETURN emp_rec;
   END nth_highest_salary;

   /* Define local function, available only in package. */
   FUNCTION rank (emp_id INTEGER, job_title VARCHAR2)
      RETURN INTEGER IS
   /* Return rank (highest = 1) of employee in a given
      job classification based on performance rating. */
      head_count INTEGER;
      score      NUMBER;
   BEGIN
      SELECT COUNT(*) INTO head_count FROM emp
         WHERE job = job_title;
      SELECT rating INTO score FROM reviews
         WHERE empno = emp_id;
      score := score / 100;  -- maximum score is 100
      RETURN (head_count + 1) - ROUND(head_count * score);
   END rank;

BEGIN  -- initialization part starts here
   INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ACTIONS');
   number_hired := 0;
END emp_actions;
```

Remember, the initialization part of a package is run just once, the first time you reference the package. So, in the last example, only one row is inserted into the database table `emp_audit`. Likewise, the variable `number_hired` is initialized only once.

Every time the procedure `hire_employee` is called, the variable `number_hired` is updated. However, the count kept by `number_hired` is session specific. That is, the count reflects the number of new employees processed by one user, *not* the number processed by all users.

In the next example, you package some typical bank transactions. Assume that debit and credit transactions are entered after business hours via automatic teller machines, then applied to accounts the next morning.

```
CREATE PACKAGE bank_transactions AS
   /* Declare externally visible constant. */
   minimum_balance  CONSTANT NUMBER := 100.00;
   /* Declare externally callable procedures. */
   PROCEDURE apply_transactions;
   PROCEDURE enter_transaction (
      acct   NUMBER,
      kind   CHAR,
      amount NUMBER);
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 NUMBER, credit REAL);
   PROCEDURE debit_account (acct NUMBER, 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   NUMBER,
   kind   CHAR,
   amount NUMBER) 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    NUMBER,
   kind    CHAR,
   new_bal NUMBER) 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 NUMBER, credit REAL) IS
/* Credit account unless account number is bad. */
   old_balance NUMBER;
   new_balance NUMBER;
```

```
   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 NUMBER, debit REAL) IS
   /* Debit account unless account number is bad or
      account has insufficient funds. */
      old_balance NUMBER;
      new_balance NUMBER;
      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

Look again at the package `emp_actions`. The package body declares a variable named `number_hired`, which is initialized to zero. Unlike items declared in the spec of `emp_actions`, items declared in the body are restricted to use within the package. Therefore, PL/SQL code outside the package cannot reference the variable `number_hired`. Such items are termed *private*.

However, items declared in the spec of `emp_actions` such as the exception `salary_missing` are visible outside the package. Therefore, any PL/SQL code can reference the exception `salary_missing`. Such items are termed *public*.

When you must maintain items throughout a session or across transactions, place them in the declarative part of the package body. For example, the value of `number_hired` is retained between calls to `hire_employee`. Remember, however, that the value of `number_hired` is session specific.

If you must also make the items public, place them in the package spec. For example, the constant `minimum_balance` declared in the spec of the package `bank_transactions` is available for general use.

Note: When you call a packaged subprogram remotely, the whole package is reinstantiated and its previous state is lost.

Overloading

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when you want a subprogram to accept parameters that have different datatypes. For example, the following package defines two procedures named `journalize`:

```
CREATE PACKAGE journal_entries AS
   ...
   PROCEDURE journalize (amount NUMBER, trans_date VARCHAR2);
   PROCEDURE journalize (amount NUMBER, trans_date NUMBER );
END journal_entries;

CREATE PACKAGE BODY journal_entries AS
   ...
   PROCEDURE journalize (amount NUMBER, trans_date VARCHAR2) IS
   BEGIN
      INSERT INTO journal
         VALUES (amount, TO_DATE(trans_date, 'DD-MON-YYYY'));
   END journalize;
```

```
   PROCEDURE journalize (amount NUMBER, trans_date NUMBER) 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). Yet, each procedure handles the data appropriately. For the rules that apply to overloaded subprograms, see "Overloading" on page 7-23.

Package STANDARD

A package named STANDARD defines the PL/SQL environment. The package spec globally declares types, exceptions, and subprograms, which are available automatically to every PL/SQL program. For example, package STANDARD declares the following built-in function named ABS, which returns the absolute value of its argument:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

The contents of package STANDARD are directly visible to applications. So, you can call ABS from database triggers, stored subprograms, 3GL applications, and various Oracle tools.

If you redeclare ABS in a PL/SQL program, your local declaration overrides the global declaration. However, you can still call the built-in function by using dot notation, as follows:

```
... STANDARD.ABS(x) ...
```

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.

Product-specific Packages

Oracle and various Oracle tools are supplied with product-specific packages that help you build PL/SQL-based applications. For example, Oracle is supplied with many utility packages, a few of which are highlighted below. For more information, see *Oracle8i Supplied Packages Reference.*

DBMS_STANDARD

Package DBMS_STANDARD provides language facilities that help your application interact with Oracle. For instance, the procedure `raise_application_error` lets you issue user-defined error messages. That way, you can report errors to an application and avoid returning unhandled exceptions. For an example, see "Using raise_application_error" on page 6-9.

DBMS_ALERT

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.

DBMS_OUTPUT

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 PROCEDURE calc_payroll (payroll OUT NUMBER) AS
   CURSOR c1 IS SELECT sal, comm FROM emp;
BEGIN
   payroll := 0;
   FOR c1rec IN c1 LOOP
      c1rec.comm := NVL(c1rec.comm, 0);
      payroll := payroll + c1rec.sal + c1rec.comm;
   END LOOP;
   /* Display debug info. */
   DBMS_OUTPUT.PUT_LINE('Value of payroll: ' || TO_CHAR(payroll));
END;
```

When you issue the following commands, SQL*Plus displays the value assigned by the procedure to parameter `payroll`:

```
SQL> SET SERVEROUTPUT ON
SQL> VARIABLE num NUMBER
SQL> CALL calc_payroll(:num);
Value of payroll: 31225
```

DBMS_PIPE

Package `DBMS_PIPE` allows different sessions to communicate over named pipes. (A *pipe* is an area of memory used by one process to pass information to another.) You can use the procedures `pack_message` and `send_message` to pack a message into a pipe, then send it to another session in the same instance.

At the other end of the pipe, you can use the procedures `receive_message` and `unpack_message` to receive and unpack (read) the message. Named pipes are useful in many ways. For example, you can write routines in C that allow external servers to collect information, then send it through pipes to procedures stored in an Oracle database.

UTL_FILE

Package `UTL_FILE` allows your PL/SQL programs to read and write operating system (OS) text files. It provides a restricted version of standard OS stream file I/O, including open, put, get, and close operations.

When you want to read or write a text file, you call the function `fopen`, which returns a file handle for use in subsequent procedure calls. For example, the procedure `put_line` writes a text string and line terminator to an open file. The procedure `get_line` reads a line of text from an open file into an output buffer.

PL/SQL file I/O is available on both the client and server sides. However, on the server side, file access is restricted to those directories explicitly listed in the *accessible directories list*, which is stored in the Oracle initialization file.

UTL_HTTP

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

When writing packages, keep them as general as possible so they can be reused in future applications. Avoid writing packages that duplicate some feature already provided by Oracle.

Package specs reflect the design of your application. So, define them before the package bodies. Place in a spec only the types, items, and subprograms that must be visible to users of the package. That way, other developers cannot misuse the package by basing their code on irrelevant implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package spec. Changes to a package body do not require Oracle to recompile dependent procedures. However, changes to a package spec require Oracle to recompile every stored subprogram that references the package.

9

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 rapidly gaining acceptance because it can reduce the cost and time required to build complex applications. In PL/SQL, object-oriented programming is based on object types. They provide abstract templates for real-world objects, and so are an ideal modeling tool. They also provide black-box encapsulation like an integrated component that can be plugged into various electronic devices. To plug an object type into your programs, you need to know only what it does, not how it works.

Major Topics

The Role of Abstraction

An *abstraction* is a high-level description or model of a real-world entity. Abstractions keep our daily lives manageable. They help us reason about an object, event, or relationship by suppressing irrelevant detail. For example, to drive a car, you need not know how its engine works. A simple interface consisting of a gearshift, steering wheel, accelerator, and brake, lets you use the car effectively. The details of what happens under the hood are not important for day-to-day driving.

Abstractions are central to the discipline of programming. For example, you use *procedural abstraction* when you suppress the details of a complex algorithm by writing a procedure and passing it parameters. A single procedure call hides the details of your implementation. To try a different implementation, you simply replace the procedure with another having the same name and parameters. Thanks to abstraction, programs that call the procedure need not be modified.

You use *data abstraction* when you specify the datatype of a variable. The datatype stipulates a set of values and a set of operations appropriate for those values. For instance, a variable of type `POSITIVE` can hold only positive integers, and can only be added, subtracted, multiplied, and so on. To use the variable, you need not know how PL/SQL stores integers or implements arithmetic operations; you simply accept the programming interface.

Object types are a generalization of the built-in datatypes found in most programming languages. PL/SQL provides a variety of built-in scalar and composite datatypes, each of which is associated with a set of predefined operations. A *scalar* type (such as `CHAR`) has no internal components. A *composite* type (such as `RECORD`) has internal components that can be manipulated individually. Like the `RECORD` type, an object type is a composite type. However, its operations are user-defined, not predefined.

Currently, you cannot define object types within PL/SQL. They must be `CREATE`d and stored in an Oracle database, where they can be shared by many programs. A program that uses object types is called a *client program*. It can declare and manipulate an object without knowing how the object type represents data or implements operations. This allows you to write the program and object type separately, and to change the implementation of the object type without affecting the program. Thus, object types support both procedural and data abstraction.

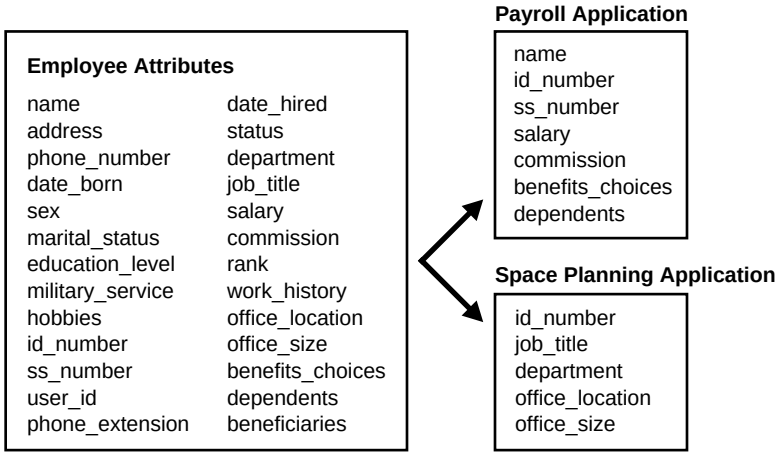
What Is an Object Type?

An *object type* is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called *attributes*. The functions and procedures that characterize the behavior of the object type are called *methods*.

We usually think of an object (such as a person, car, or bank account) as having attributes and behaviors. For example, a baby has the attributes gender, age, and weight, and the behaviors eat, drink, and sleep. Object types let you maintain this perspective when you sit down to write an application.

When you define an object type using the CREATE TYPE statement, you create an abstract template for some real-world object. The template specifies only those attributes and behaviors the object will need in the application environment. For example, an employee has many attributes, but usually only a few are needed to fill the requirements of an application (see Figure 9–1).

Figure 9–1 Form Follows Function



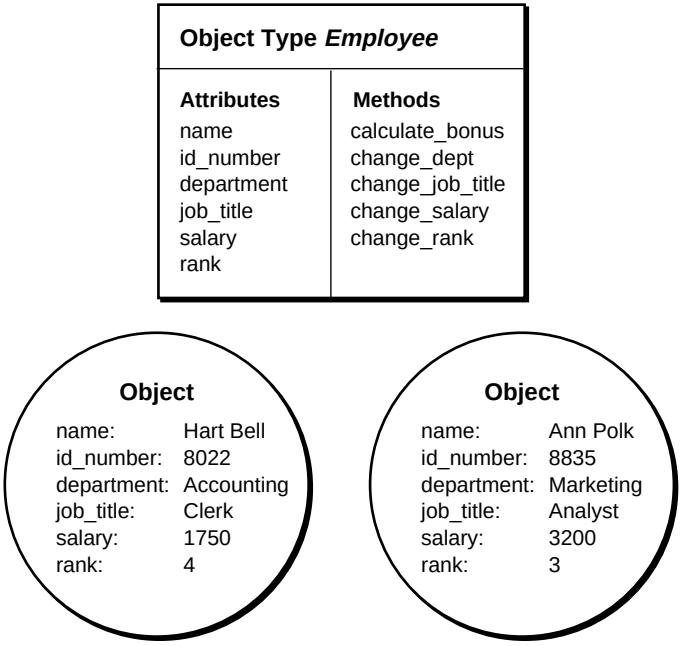
Suppose you must write a program to allocate employee bonuses. Not all employee attributes are needed to solve this problem. So, you design an abstract employee who has the following problem-specific attributes: name, id_number, department, job title, salary, and rank. Then, you identify the operations needed to handle an abstract employee. For example, you need an operation that lets Management change the rank of an employee.

Next, you define a set of variables (attributes) to represent the data, and a set of subprograms (methods) to perform the operations. Finally, you encapsulate the attributes and methods in an object type.

The data structure formed by the set of attributes is public (visible to client programs). However, well-behaved programs do not manipulate it directly. Instead, they use the set of methods provided. That way, the employee data is kept in a proper state.

At run time, when the data structure is filled with values, you have created an *instance* of an abstract employee. You can create as many instances (usually called *objects*) as you need. Each object has the name, number, job title, and so on of an actual employee (see Figure 9–2). This data is accessed or changed only by the methods associated with it. Thus, object types let you create objects with well-defined attributes and behavior.

Figure 9–2 Object Type and Objects (Instances) of That Type



Why Use Object Types?

Object types reduce complexity by breaking down a large system into logical entities. This allows you to create software components that are modular, maintainable, and reusable. It also allows different teams of programmers to develop software components concurrently.

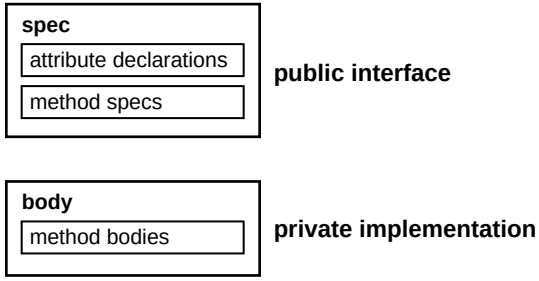
By encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods. Object types minimize side effects by allowing access to data only through approved operations. Also, object types hide implementation details, so that you can change the details without affecting client programs.

Object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. Moreover, object types map directly into classes defined in object-oriented languages such as Java and C++. Now your programs can better reflect the world they are trying to simulate.

Structure of an Object Type

Like a package, an object type has two parts: a specification and a body (refer to Figure 9–3). 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.

Figure 9–3 Object Type Structure



All the information a client program needs to use the methods is in the spec. Think of the spec as an operational interface and of the body as a black box. You can debug, enhance, or replace the body without changing the spec—and without affecting client programs.

In an object type spec, all attributes must be declared before any methods. Only subprograms have an underlying implementation. So, if an object type spec declares only attributes, the object type body is unnecessary. You cannot declare attributes in the body. All declarations in the object type spec are public (visible outside the object type).

To understand the structure better, study the example below, in which an object type for complex numbers is defined. For now, it is enough to know that a complex number has two parts, a real part and an imaginary part, and that several arithmetic operations are defined for complex numbers.

```
CREATE TYPE Complex AS OBJECT (
   rpart REAL,  -- attribute
   ipart REAL,
   MEMBER FUNCTION plus (x Complex) RETURN Complex,  -- method
   MEMBER FUNCTION less (x Complex) RETURN Complex,
   MEMBER FUNCTION times (x Complex) RETURN Complex,
   MEMBER FUNCTION divby (x Complex) RETURN Complex
);

CREATE TYPE BODY Complex AS
   MEMBER FUNCTION plus (x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart + x.rpart, ipart + x.ipart);
   END plus;

   MEMBER FUNCTION less (x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart - x.rpart, ipart - x.ipart);
   END less;

   MEMBER FUNCTION times (x Complex) RETURN Complex IS
   BEGIN
      RETURN Complex(rpart * x.rpart - ipart * x.ipart,
                     rpart * x.ipart + ipart * x.rpart);
   END times;

   MEMBER FUNCTION divby (x Complex) RETURN Complex IS
      z REAL := x.rpart**2 + x.ipart**2;
   BEGIN
      RETURN Complex((rpart * x.rpart + ipart * x.ipart) / z,
                     (ipart * x.rpart - rpart * x.ipart) / z);
   END divby;
END;
```

Components of an Object Type

An object type encapsulates data and operations. So, 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. You *cannot* add attributes to an existing object type (that is, type evolution is not supported).

Attributes

Like a variable, an attribute is declared with a name and datatype. The name must be unique within the object type (but can be reused in other object types). The datatype can be any Oracle type except

- LONG and LONG RAW
- NCHAR, NCLOB, and NVARCHAR2
- ROWID and UROWID
- the PL/SQL-specific types BINARY_INTEGER (and its subtypes), BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE, and %ROWTYPE
- types defined inside a PL/SQL package

You cannot initialize an attribute in its declaration using the assignment operator or DEFAULT clause. Also, you cannot impose the NOT NULL constraint on an attribute. However, objects can be stored in database tables on which you can impose constraints.

The kind of data structure formed by a set of attributes depends on the real-world object being modeled. For example, to represent a rational number, which has a numerator and a denominator, you need only two INTEGER variables. On the other hand, to represent a college student, you need several VARCHAR2 variables to hold a name, address, phone number, status, and so on, plus a VARRAY variable to hold courses and grades.

The data structure can be very complex. For example, the datatype of an attribute can be another object type (called a *nested* object type). That lets you build a complex object type from simpler object types. Some object types such as queues, lists, and trees are dynamic, meaning that they can grow as they are used. Recursive object types, which contain direct or indirect references to themselves, allow for highly sophisticated data models.

Methods

In general, a method is a subprogram declared in an object type spec using the keyword MEMBER or STATIC. The method cannot have the same name as the object type or any of its attributes. MEMBER methods are invoked on instances, as in

```
instance_expression.method()
```

However, STATIC methods are invoked on the object type, not its instances, as in

```
object_type_name.method()
```

Like packaged subprograms, most 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 be a corresponding method body in the object type body. To match method specs and bodies, the PL/SQL compiler does a token-by-token comparison of their headers. So, the headers must match word for word.

In an object type, methods can reference attributes and other methods without a qualifier, as the following example shows:

```
CREATE TYPE Stack AS OBJECT (
   top INTEGER,
   MEMBER FUNCTION full RETURN BOOLEAN,
   MEMBER PROCEDURE push (n IN INTEGER),
   ...
);

CREATE TYPE BODY Stack AS
   ...
   MEMBER PROCEDURE push (n IN INTEGER) IS
   BEGIN
      IF NOT full THEN
         top := top + 1;
         ...
   END push;
END;
```

Parameter SELF

MEMBER methods accept a built-in parameter named SELF, which is an instance of the object type. Whether declared implicitly or explicitly, it is always the first parameter passed to a MEMBER method. However, STATIC methods cannot accept or reference SELF.

In the method body, SELF denotes the object whose method was invoked. For example, method transform declares SELF as an IN OUT parameter:

```
CREATE TYPE Complex AS OBJECT (
   MEMBER FUNCTION transform (SELF IN OUT Complex) ...
```

You cannot specify a different datatype for SELF. In MEMBER functions, if SELF is not declared, its parameter mode defaults to IN. However, in MEMBER procedures, if SELF is not declared, its parameter mode defaults to IN OUT. You cannot specify the OUT parameter mode for SELF.

As the following example shows, methods can reference the attributes of SELF without a qualifier:

```
CREATE 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. That is, you can use the same name for different methods if their formal parameters differ in number, order, or datatype family. When you call one of the methods, PL/SQL finds it by comparing the list of actual parameters with each list of formal parameters.

You cannot overload two methods if their formal parameters differ only in parameter mode. Also, you cannot overload two member functions that differ only in return type. For more information, see "Overloading" on page 7-23.

Map and Order Methods

The values of a scalar datatype such as CHAR or REAL have a predefined order, which allows them to be compared. But, instances of an object type have no predefined order. To put them in order, 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 ordering to evaluate Boolean expressions such as `x > y`, and to do comparisons implied by the `DISTINCT`, `GROUP BY`, and `ORDER BY` clauses. Map method `convert` returns the relative position of an object in the ordering of all `Rational` objects.

An object type can contain only one map method, which must be a parameterless function with one of the following scalar return 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 just two parameters: the built-in parameter `SELF` and another object of the same type. If `c1` and `c2` are `Customer` objects, a comparison such as `c1 > c2` calls method `match` automatically. The method returns a negative number, zero, or a positive number signifying that `SELF` is respectively less than, equal to, or greater than the other parameter. If either parameter passed to an order method is null, the method returns a null.

Guidelines A map method, acting like a hash function, maps object values into scalar values (which are easier to compare), then compares the scalar values. An order method simply compares one object value to another.

You can declare a map method or an order method but not both. If you declare either method, you can compare objects in SQL and procedural statements. However, if you declare neither method, you can compare objects only in SQL statements and only for equality or inequality. (Two objects of the same type are equal *only if* the values of their corresponding attributes are equal.)

When sorting or merging a large number of objects, use a map method. One call maps all the objects into scalars, then sorts the scalars. An order method is less efficient because it must be called repeatedly (it can compare only two objects at a time). You must use a map method for hash joins because PL/SQL hashes on the object value.

Constructor Methods

Every object type has a *constructor method* (*constructor* for short), which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type.

Oracle generates a default constructor for every object type. The formal parameters of the constructor match the attributes of the object type. That is, the parameters and attributes are declared in the same order and have the same names and datatypes. PL/SQL never calls a constructor implicitly, so you must call it explicitly. For more information, see "Calling Constructors" on page 9-25.

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. However, you can define them interactively in SQL*Plus using the following syntax:

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT (
  attribute_name datatype[, attribute_name datatype]...
  [{MAP | ORDER} MEMBER function_spec,]
  [{MEMBER | STATIC} {subprogram_spec | call_spec}
  [, {MEMBER | STATIC} {subprogram_spec | call_spec}]...]
);
```

```
[CREATE [OR REPLACE] TYPE BODY type_name {IS | AS}
  { {MAP | ORDER} MEMBER function_body;
   | {MEMBER | STATIC} {subprogram_body | call_spec};}
  [{MEMBER | STATIC} {subprogram_body | call_spec};]...
END;]
```

The AUTHID clause determines whether all member methods execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights versus Definer Rights" on page 7-29.

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 *Oracle8i Java Stored Procedures Developer's Guide*. To learn how to write C call specs, see *Oracle8i Application Developer's Guide - Fundamentals*.

Object Type *Stack*

A *stack* holds an ordered collection of data items. As the name implies, stacks have a top and a bottom. But, items can be added or removed only at the top. So, the last item added to a stack is the first item removed. (Think of the stack of clean serving trays in a cafeteria.) The operations *push* and *pop* update the stack while preserving last in, first out (LIFO) behavior.

Stacks have many applications. For example, they are used in systems programming to prioritize interrupts and to manage recursion. The simplest implementation of a stack uses an integer array. Integers are stored in array elements, with one end of the array representing the top of the stack.

PL/SQL provides the datatype VARRAY, which allows you to declare variable-size arrays (varrays for short). To declare a varray attribute, you must first define its type. However, you cannot define types in an object type spec. So, you define a stand-alone varray type, specifying its maximum size, as follows:

```
CREATE TYPE IntArray AS VARRAY(25) OF INTEGER;
```

Now, you can write the object type spec:

```
CREATE TYPE Stack AS OBJECT (
   max_size INTEGER,
   top      INTEGER,
   position IntArray,
   MEMBER PROCEDURE initialize,
   MEMBER FUNCTION full RETURN BOOLEAN,
   MEMBER FUNCTION empty RETURN BOOLEAN,
   MEMBER PROCEDURE push (n IN INTEGER),
   MEMBER PROCEDURE pop (n OUT INTEGER)
);
```

Finally, you write the object type body:

```
CREATE TYPE BODY Stack AS
   MEMBER PROCEDURE initialize IS
   BEGIN
      top := 0;
      /* Call constructor for varray and set element 1 to NULL. */
      position := IntArray(NULL);
      max_size := position.LIMIT;  -- get varray size constraint
      position.EXTEND(max_size - 1, 1); -- copy element 1 into 2..25
   END initialize;

   MEMBER FUNCTION full RETURN BOOLEAN IS
   BEGIN
      RETURN (top = max_size);  -- return TRUE if stack is full
   END full;

   MEMBER FUNCTION empty RETURN BOOLEAN IS
   BEGIN
      RETURN (top = 0);  -- return TRUE if stack is empty
   END empty;

   MEMBER PROCEDURE push (n IN INTEGER) IS
   BEGIN
      IF NOT full THEN
         top := top + 1;  -- push integer onto stack
         position(top) := n;
      ELSE  -- stack is full
         RAISE_APPLICATION_ERROR(-20101, 'stack overflow');
      END IF;
   END push;
```

```
   MEMBER PROCEDURE pop (n OUT INTEGER) IS
   BEGIN
      IF NOT empty THEN
         n := position(top);
         top := top - 1;  -- pop integer off stack
      ELSE  -- stack is empty
         RAISE_APPLICATION_ERROR(-20102, 'stack underflow');
      END IF;
   END pop;
END;
```

In member procedures push and pop, you use the built-in procedure raise_application_error to issue user-defined error messages. That way, you report errors to the client program and avoid returning unhandled exceptions to the host environment. The client program gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS exception handler, as follows:

```
DECLARE
   err_num NUMBER;
   err_msg VARCHAR2(100);
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN
      err_num := SQLCODE;
      err_msg := SUBSTR(SQLERRM, 1, 100);
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(err_num) || ': ' || err_msg);
```

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.

Alternatively, the program can use pragma EXCEPTION_INIT to map the error numbers returned by raise_application_error to named exceptions, as the following example shows:

```
DECLARE
   stack_overflow  EXCEPTION;
   stack_underflow EXCEPTION;
   PRAGMA EXCEPTION_INIT(stack_overflow, -20101);
   PRAGMA EXCEPTION_INIT(stack_underflow, -20102);
BEGIN
   ...
EXCEPTION
   WHEN stack_overflow THEN ...
```

Object Type *Ticket_Booth*

Consider a chain of low-budget, triplex movie theatres. Each theatre has a ticket booth where tickets for three different movies are sold. All tickets are priced at $3.00. Periodically, ticket receipts are collected and the stock of tickets is replenished.

Before defining an object type that represents a ticket booth, you must consider the data and operations needed. For a simple ticket booth, the object type needs attributes for the ticket price, quantity of tickets on hand, and receipts. It also needs methods for the following operations: purchase ticket, take inventory, replenish stock, and collect receipts.

For receipts, you use a three-element varray. Elements 1, 2, and 3 record the ticket receipts for movies 1, 2, and 3, respectively. To declare a varray attribute, you must first define its type, as follows:

```
CREATE TYPE RealArray AS VARRAY(3) OF REAL;
```

Now, you can write the object type spec:

```
CREATE TYPE Ticket_Booth AS OBJECT (
   price       REAL,
   qty_on_hand INTEGER,
   receipts    RealArray,
   MEMBER PROCEDURE initialize,
   MEMBER PROCEDURE purchase (
      movie  INTEGER,
      amount REAL,
      change OUT REAL),
   MEMBER FUNCTION inventory RETURN INTEGER,
   MEMBER PROCEDURE replenish (quantity INTEGER),
   MEMBER PROCEDURE collect (movie INTEGER, amount OUT REAL)
);
```

Finally, you write the object type body:

```
CREATE TYPE BODY Ticket_Booth AS
   MEMBER PROCEDURE initialize IS
   BEGIN
      price := 3.00;
      qty_on_hand := 5000;  -- provide initial stock of tickets
      -- call constructor for varray and set elements 1..3 to zero
      receipts := RealArray(0,0,0);
   END initialize;
```

```
   MEMBER PROCEDURE purchase (
      movie  INTEGER,
      amount REAL,
      change OUT REAL) IS
   BEGIN
      IF qty_on_hand = 0 THEN
         RAISE_APPLICATION_ERROR(-20103, 'out of stock');
      END IF;
      IF amount >= price THEN
         qty_on_hand := qty_on_hand - 1;
         receipts(movie) := receipts(movie) + price;
         change := amount - price;
      ELSE  -- amount is not enough
         change := amount;  -- so return full amount
      END IF;
   END purchase;

   MEMBER FUNCTION inventory RETURN INTEGER IS
   BEGIN
      RETURN qty_on_hand;
   END inventory;

   MEMBER PROCEDURE replenish (quantity INTEGER) IS
   BEGIN
      qty_on_hand := qty_on_hand + quantity;
   END replenish;

   MEMBER PROCEDURE collect (movie INTEGER, amount OUT REAL) IS
   BEGIN
      amount := receipts(movie);  -- get receipts for a given movie
      receipts(movie) := 0;  -- reset receipts to zero
   END collect;
END;
```

Object Type *Bank_Account*

Before defining an object type that represents a bank account, you must consider the data and operations needed. For a simple bank account, the object type needs attributes for an account number, balance, and status. It also needs methods for the following operations: open account, verify account number, close account, deposit money, withdraw money, and return balance.

First, you write the object type spec, as follows:

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

Then, you write the object type body:

```
CREATE TYPE BODY Bank_Account AS
   MEMBER PROCEDURE open (amount IN REAL) IS
   -- open account with initial deposit
   BEGIN
      IF NOT amount > 0 THEN
         RAISE_APPLICATION_ERROR(-20104, 'bad amount');
      END IF;
      SELECT acct_sequence.NEXTVAL INTO acct_number FROM dual;
      status := 'open';
      balance := amount;
   END open;

   MEMBER PROCEDURE verify_acct (num IN INTEGER) IS
   -- check for wrong account number or closed account
   BEGIN
      IF (num <> acct_number) THEN
         RAISE_APPLICATION_ERROR(-20105, 'wrong number');
      ELSIF (status = 'closed') THEN
         RAISE_APPLICATION_ERROR(-20106, 'account closed');
      END IF;
   END verify_acct;
```

```
   MEMBER PROCEDURE close (num IN INTEGER, amount OUT REAL) IS
   -- close account and return balance
   BEGIN
      verify_acct(num);
      status := 'closed';
      amount := balance;
   END close;

   MEMBER PROCEDURE deposit (num IN INTEGER, amount IN REAL) IS
   BEGIN
      verify_acct(num);
      IF NOT amount > 0 THEN
         RAISE_APPLICATION_ERROR(-20104, 'bad amount');
      END IF;
      balance := balance + amount;
   END deposit;

   MEMBER PROCEDURE withdraw (num IN INTEGER, amount IN REAL) IS
   -- if account has enough funds, withdraw
   -- given amount; else, raise an exception
   BEGIN
      verify_acct(num);
      IF amount <= balance THEN
         balance := balance - amount;
      ELSE
         RAISE_APPLICATION_ERROR(-20107, 'insufficient funds');
      END IF;
   END withdraw;

   MEMBER FUNCTION curr_bal (SELF IN OUT Bank_Account, num IN
INTEGER)
      RETURN REAL IS
   BEGIN
      verify_acct(num);
      RETURN balance;
   END curr_bal;
END;
```

Object Type *Rational*

A rational number is a number expressible as the quotient of two integers, a numerator and a denominator. Like most languages, PL/SQL does not have a rational number type or predefined operations on rational numbers. Let us remedy that omission by defining object type `Rational`. First, you write the object type spec, as follows:

```
CREATE TYPE Rational AS OBJECT (
   num INTEGER,
   den INTEGER,
   MAP MEMBER FUNCTION convert RETURN REAL,
   MEMBER PROCEDURE normalize,
   MEMBER FUNCTION reciprocal RETURN Rational,
   MEMBER FUNCTION plus (x Rational) RETURN Rational,
   MEMBER FUNCTION less (x Rational) RETURN Rational,
   MEMBER FUNCTION times (x Rational) RETURN Rational,
   MEMBER FUNCTION divby (x Rational) RETURN Rational,
   PRAGMA RESTRICT_REFERENCES (DEFAULT, RNDS,WNDS,RNPS,WNPS)
);
```

PL/SQL does not allow the overloading of operators. So, you must define methods named `plus`, `less` (the word `minus` is reserved), `times`, and `divby` instead of overloading the infix operators +, –, *, and /.

Next, you create the following stand-alone stored function, which will be called by method `normalize`:

```
CREATE FUNCTION gcd (x INTEGER, y INTEGER) RETURN INTEGER AS
-- find greatest common divisor of x and y
   ans INTEGER;
BEGIN
   IF (y <= x) AND (x MOD y = 0) THEN
      ans := y;
   ELSIF x < y THEN
      ans := gcd(y, x);  -- recursive call
   ELSE
      ans := gcd(y, x MOD y);  -- recursive call
   END IF;
   RETURN ans;
END;
```

Then, you write the object type body, as follows:

```
CREATE TYPE BODY Rational AS
   MAP MEMBER FUNCTION convert RETURN REAL IS
   -- convert rational number to real number
   BEGIN
      RETURN num / den;
   END convert;

   MEMBER PROCEDURE normalize IS
   -- reduce fraction num / den to lowest terms
      g INTEGER;
   BEGIN
      g := gcd(num, den);
      num := num / g;
      den := den / g;
   END normalize;

   MEMBER FUNCTION reciprocal RETURN Rational IS
   -- return reciprocal of num / den
   BEGIN
      RETURN Rational(den, num);  -- call constructor
   END reciprocal;

   MEMBER FUNCTION plus (x Rational) RETURN Rational IS
   -- return sum of SELF + x
      r Rational;
   BEGIN
      r := Rational(num * x.den + x.num * den, den * x.den);
      r.normalize;
      RETURN r;
   END plus;

   MEMBER FUNCTION less (x Rational) RETURN Rational IS
   -- return difference of SELF - x
      r Rational;
   BEGIN
      r := Rational(num * x.den - x.num * den, den * x.den);
      r.normalize;
      RETURN r;
   END less;

   MEMBER FUNCTION times (x Rational) RETURN Rational IS
   -- return product of SELF * x
      r Rational;
```

```
   BEGIN
      r := Rational(num * x.num, den * x.den);
      r.normalize;
      RETURN r;
   END times;

   MEMBER FUNCTION divby (x Rational) RETURN Rational IS
   -- return quotient of SELF / x
      r Rational;
   BEGIN
      r := Rational(num * x.den, den * x.num);
      r.normalize;
      RETURN r;
   END divby;
END;
```

Declaring and Initializing Objects

Once an object type is defined and installed in the schema, you can use it to declare objects in any PL/SQL block, subprogram, or package. For example, you can use the object type to specify the datatype of an attribute, column, variable, bind variable, record field, table element, formal parameter, or function result. At 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 are allowed, in which case SELF is bound to NULL. When passed as arguments to IN parameters, attributes of an uninitialized object evaluate to NULL. When passed as arguments to OUT or IN OUT parameters, they raise an exception if you try to write to them.

Accessing Attributes

You can refer to an attribute only by name (not by its position in the object type). To access or change the value of an attribute, you use dot notation. In the example below, you assign the value of attribute den to variable denominator. Then, you assign the value stored in variable numerator to attribute num.

```
DECLARE
   r Rational := Rational(NULL, NULL);
   numerator   INTEGER;
   denominator INTEGER;
BEGIN
   ...
   denominator := r.den;
   r.num := numerator;
```

Attribute names can be chained, which allows you to 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       VARCAHR2(10),
   advisor_name VARCHAR2(20),
   ...
);'
```

Notice that `zip_code` is an attribute of object type `Address` and that `Address` is the datatype of attribute `home_address` in object type `Student`. If `s` is a `Student` object, you access the value of its `zip_code` attribute as follows:

```
s.home_address.zip_code
```

Calling Constructors

Calls to a constructor are allowed wherever function calls are allowed. Like all functions, a constructor is called as part of an expression, as the following example shows:

```
DECLARE
   r1 Rational := Rational(2, 3);
   FUNCTION average (x Rational, y Rational) RETURN Rational IS
   BEGIN
      ...
   END;
BEGIN
   r1 := average(Rational(3, 4), Rational(7, 11));
   IF (Rational(5, 8) > r1) THEN
      ...
   END IF;
END;
```

When you pass parameters to a constructor, the call assigns initial values to the attributes of the object being instantiated. You must supply a parameter for every attribute because, unlike constants and variables, attributes cannot have default values. As the following example shows, the *n*th parameter assigns a value to the *n*th attribute:

```
DECLARE
   r Rational;
BEGIN
   r := Rational(5, 6);  -- assign 5 to num, 6 to den
   -- now r is 5/6
```

The next example shows that you can call a constructor using named notation instead of positional notation:

```
BEGIN
   r := Rational(den => 6, num => 5);  -- assign 5 to num, 6 to den
```

Calling Methods

Like packaged subprograms, methods are called using dot notation. In the following example, you call method `normalize`, which divides attributes `num` and `den` by their greatest common divisor:

```
DECLARE
   r Rational;
BEGIN
   r := Rational(6, 8);
   r.normalize;
   DBMS_OUTPUT.PUT_LINE(r.num);  -- prints 3
```

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
```

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 illegal:

```
r.normalize().reciprocal;  -- illegal
```

Also, if you chain two function calls, the first function must return an object that can be passed to the second function.

Sharing Objects

Most real-world objects are considerably larger and more complex than objects of type `Rational`. Consider the following 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,
   ...
);
```

`Address` objects have twice as many attributes as `Rational` objects, and `Person` objects have still more attributes including one of type `Address`. When an object is large, it is inefficient to pass copies of it from subprogram to subprogram. It makes more sense to share the object. You can do that if the object has an object identifier. To share the object, you use references (refs for short). A *ref* is a pointer to an object.

Sharing has two important advantages. First, data is not replicated unnecessarily. Second, when a shared object is updated, the change occurs in only one place, and any ref can retrieve the updated values instantly.

In the following example, you gain the advantages of sharing by defining object type Home and then creating a table that stores instances of that object type:

```
CREATE TYPE Home AS OBJECT (
   address    VARCHAR2(35),
   owner      VARCHAR2(25),
   age        INTEGER,
   style      VARCHAR(15),
   floor plan BLOB,
   price      REAL(9,2),
   ...
);
...
CREATE TABLE homes OF Home;
```

Using Refs

By revising object type Person, you can model a community in which several people might share the same home. You use the type modifier REF to declare refs, which hold pointers to objects.

```
CREATE TYPE Person AS OBJECT (
   first_name   VARCHAR2(10),
   last_name    VARCHAR2(15),
   birthday     DATE,
   home_address REF Home,  -- can be shared by family
   phone_number VARCHAR2(15),
   ss_number    INTEGER,
   mother       REF Person, -- family members refer to each other
   father       REF Person,
   ...
);
```

Notice how references from persons to homes and between persons model real-world relationships.

You can declare refs as variables, parameters, fields, or attributes. And, you can use refs as input or output variables in SQL data manipulation statements. However, you cannot navigate through refs. Given an expression such as x.attribute, where x is a ref, PL/SQL cannot navigate to the table in which the referenced object is stored. For example, the following assignment is illegal:

```
DECLARE
   p_ref    REF Person;
   phone_no VARCHAR2(15);
```

```
BEGIN
   phone_no := p_ref.phone_number;  -- illegal
```

Instead, you must use the operator DEREF to access the object. For some examples, see "Using Operator DEREF" on page 9-33.

Forward Type Definitions

You can refer only to schema objects that already exist. In the following example, the first CREATE TYPE statement is illegal because it refers to object type Department, which does not yet exist:

```
CREATE TYPE Employee AS OBJECT (
   name VARCHAR2(20),
   dept REF Department,  -- illegal
   ...
);

CREATE TYPE Department AS OBJECT (
   number  INTEGER,
   manager Employee,
   ...
);
```

Switching the CREATE TYPE statements does not help because the object types are *mutually dependent*; that is, one depends on the other through a ref. 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 allows you to defer the definition of object type `Address`. Moreover, the incomplete type `Customer` can be made available to other application developers for use in refs.

Manipulating Objects

You can use an object type in the `CREATE TABLE` statement to specify the datatype of a column. Once the table is created, you can use SQL statements to insert an object, select its attributes, call its methods, and update its state.

Note: Access to remote or distributed objects is not allowed.

In the SQL*Plus script below, the `INSERT` statement calls the constructor for object type `Rational`, then inserts the resulting object. The `SELECT` statement retrieves the value of attribute `num`. The `UPDATE` statement calls member method `reciprocal`, which returns a `Rational` value after swapping attributes `num` and `den`. Notice that a table alias is required when you reference an attribute or method. (For an explanation, see Appendix D.)

```
CREATE TABLE numbers (rn Rational, ...)
/
INSERT INTO numbers (rn) VALUES (Rational(3, 62))  -- inserts 3/62
/
SELECT n.rn.num INTO my_num FROM numbers n ...  -- returns 3
/
UPDATE numbers n SET n.rn = n.rn.reciprocal ...  -- yields 62/3
/
```

When you instantiate an object this way, it has no identity outside the database table. However, the object type exists independently of any table, and can be used to create objects in other ways.

In the next example, you create a table that stores objects of type `Rational` in its rows. Such tables, having rows of objects, are called *object tables*. Each column in a row corresponds to an attribute of the object type. Rows can have different column values.

```
CREATE TABLE rational_nums OF Rational;
```

Each row in an object table has an *object identifier*, which uniquely identifies the object stored in that row and serves as a reference to the object.

Selecting Objects

Assume that you have run the following SQL*Plus script, which creates object type Person and object table persons, and that you have populated the table:

```
CREATE TYPE Person AS OBJECT (
   first_name   VARCHAR2(15),
   last_name    VARCHAR2(15),
   birthday     DATE,
   home_address Address,
   phone_number VARCHAR2(15))
/
CREATE TABLE persons OF Person
/
```

The following subquery produces a result set of rows containing only the attributes of Person objects:

```
BEGIN
   INSERT INTO employees  -- another object table of type Person
      SELECT * FROM persons p WHERE p.last_name LIKE '%Smith';
```

To return a result set of objects, you must use the operator VALUE, which is discussed in the next section.

Using Operator VALUE

As you might expect, the operator 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 Operator REF

You can retrieve refs using the operator 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';
```

In the next example, you retrieve a ref and attribute at the same time:

```
DECLARE
   p_ref       REF Person;
   taxpayer_id VARCHAR2(9);
BEGIN
   SELECT REF(p), p.ss_number INTO p_ref, taxpayer_id
      FROM persons p
      WHERE p.last_name = 'Parker'; -- must return one row
   ...
END;
```

In the final example, you update the attributes of a Person object:

```
DECLARE
   p_ref        REF Person;
   my_last_name VARCHAR2(15);
   ...
```

```
BEGIN
   ...
   SELECT REF(p) INTO p_ref FROM persons p
      WHERE p.last_name = my_last_name;
   UPDATE persons p
      SET p = Person('Jill', 'Anders', '11-NOV-67', ...)
      WHERE REF(p) = p_ref;
END;
```

Testing for Dangling Refs

If the object to which a ref points is deleted, the ref is left *dangling* (pointing to a nonexistent object). To test for this condition, you can use the SQL predicate `IS DANGLING`. For example, suppose column `manager` in relational table `department` holds refs to `Employee` objects stored in an object table. You can use the following `UPDATE` statement to convert any dangling refs into nulls:

```
BEGIN
   UPDATE department SET manager = NULL WHERE manager IS DANGLING;
```

Using Operator DEREF

You cannot navigate through refs within PL/SQL procedural statements. Instead, you must use the operator `DEREF` in a SQL statement. (`DEREF` is short for dereference. When you *dereference* a pointer, you get the value to which it points.) `DEREF` takes as its argument a reference to an object, then returns the value of that object. If the ref is dangling, `DEREF` returns a null object.

In the example below, you dereference a ref to a `Person` object. Notice that you select the ref from dummy table `dual`. You need not specify an object table and search criteria because each object stored in an object table has a unique, immutable object identifier, which is part of every ref to that object.

```
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, as the following example shows:

```
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 operator DEREF within procedural statements:

```
BEGIN
   ...
   p1 := DEREF(p_ref);  -- illegal
```

Within SQL statements, you can use dot notation to navigate through object columns to ref attributes and through one ref attribute to another. You can also navigate through ref columns to attributes if you use a table alias. For example, the following syntax is valid:

```
table_alias.object_column.ref_attribute
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, as follows:

```
DECLARE
   addr1 Address,
   addr2 Address,
   ...
BEGIN
   SELECT DEREF(home_address) INTO addr1 FROM persons p
      WHERE p.last_name = 'Derringer';
```

In the example below, you navigate through ref column `home_address` to attribute `street`. In this case, a table alias is required.

```
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, you use the `WHERE` clause, as follows:

```
BEGIN
   DELETE FROM persons p
      WHERE p.home_address = '108 Palm Dr';
   ...
END;
```

10

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 applications more flexible and versatile. You learn simple ways to write programs that can build and process SQL statements "on the fly" at run time.

Within PL/SQL, you can execute any kind of SQL statement (even data definition and data control statements) without resorting to cumbersome programmatic approaches. Dynamic SQL blends seamlessly into your programs, making them more efficient, readable, and concise.

Major Topics

What Is Dynamic SQL?

Most PL/SQL programs do a specific, predictable job. For example, a stored procedure might accept an employee number and salary increase, then update the `sal` column in the `emp` table. In this case, the full text of the `UPDATE` statement is known at compile time. Such statements do not change from execution to execution. So, they are called *static* SQL statements.

However, some programs must build and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different `SELECT` statements for the various reports it generates. In this case, the full text of the statement is unknown until run time. Such statements can, and probably will, change from execution to execution. So, they are called *dynamic* SQL statements.

Dynamic SQL statements are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments. A *placeholder* is an undeclared identifier, so its name, to which you must prefix a colon, does not matter. For example, PL/SQL makes no distinction between the following strings:

```
'DELETE FROM emp WHERE sal > :my_sal AND comm < :my_comm'
'DELETE FROM emp WHERE sal > :s AND comm < :c'
```

To process most dynamic SQL statements, you use the `EXECUTE IMMEDIATE` statement. However, to process a multi-row query (`SELECT` statement), you must use the `OPEN-FOR`, `FETCH`, and `CLOSE` statements.

The Need for Dynamic SQL

You need dynamic SQL in the following situations:

- You want to execute a SQL data definition statement (such as `CREATE`), a data control statement (such as `GRANT`), or a session control statement (such as `ALTER SESSION`). In PL/SQL, such statements cannot be executed statically.
- You want more flexibility. For example, you might want to defer your choice of schema objects until run time. Or, you might want your program to build different search conditions for the `WHERE` clause of a `SELECT` statement. A more complex program might choose from various SQL operations, clauses, etc.
- You use package `DBMS_SQL` to execute SQL statements dynamically, but you want better performance, something easier to use, or functionality that `DBMS_SQL` lacks such as support for objects and collections. (For a comparison with `DBMS_SQL`, see *Oracle8i Application Developer's Guide - Fundamentals*.)

Using the EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes a dynamic SQL statement or an anonymous PL/SQL block. The syntax is

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
    [, [IN | OUT | IN OUT] bind_argument]...];
```

where dynamic_string is a string expression that represents a SQL statement or PL/SQL block, define_variable is a variable that stores a SELECTed column value, record is a user-defined or %ROWTYPE record that stores a SELECTed row, and bind_argument is an expression whose value is passed to the dynamic SQL statement or PL/SQL block. (The NOCOPY compiler hint is not allowed in an EXECUTE IMMEDIATE statement.)

Except for multi-row queries, the string can contain any SQL statement (*without* the terminator) or any PL/SQL block (with the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement. For the right way, see "Passing the Names of Schema Objects" on page 10-10.

The INTO clause, useful only for single-row queries, specifies the variables or record into which column values are fetched. For each column value returned by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.

You must put every bind argument in the USING clause. If you do not specify a parameter mode, it defaults to IN. At run time, any bind arguments in the USING clause replace corresponding placeholders in the SQL statement or PL/SQL block. So, every placeholder must be associated with a bind argument in the USING clause. Numeric, character, and string literals are allowed in the USING clause, but Boolean literals (TRUE, FALSE, NULL) are not. To pass nulls to the dynamic string, you must use a workaround (see "Passing Nulls" on page 10-12).

Dynamic SQL supports all the SQL datatypes. So, for example, define variables and bind arguments can be collections, LOBs, instances of an object type, and REFs. As a rule, dynamic SQL does not support PL/SQL-specific types. So, for example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the INTO clause.

Caution: You can execute a dynamic SQL statement repeatedly using new values for the bind arguments. However, you incur some overhead because EXECUTE IMMEDIATE re-prepares the dynamic string before every execution.

Some Examples

The following PL/SQL block contains several examples of dynamic SQL:

```
DECLARE
   sql_stmt    VARCHAR2(100);
   plsql_block VARCHAR2(200);
   my_deptno   NUMBER(2) := 50;
   my_dname    VARCHAR2(15) := 'PERSONNEL';
   my_loc      VARCHAR2(15) := 'DALLAS';
   emp_rec     emp%ROWTYPE;
BEGIN
   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING my_deptno, my_dname, my_loc;

   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING 7788;

   EXECUTE IMMEDIATE 'DELETE FROM dept
      WHERE deptno = :n' USING my_deptno;

   plsql_block := 'BEGIN emp_stuff.raise_salary(:id, :amt); END;';
   EXECUTE IMMEDIATE plsql_block USING 7788, 500;

   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

   sql_stmt := 'ALTER SESSION SET SQL_TRACE TRUE';
   EXECUTE IMMEDIATE sql_stmt;
END;
```

In the example below, a stand-alone procedure accepts the name of a database table (such as `'emp'`) and an optional `WHERE`-clause condition (such as `'sal > 2000'`). If you omit the condition, the procedure deletes all rows from the table. Otherwise, the procedure deletes only those rows that meet the condition.

```
CREATE PROCEDURE delete_rows (
   table_name IN VARCHAR2,
   condition IN VARCHAR2 DEFAULT NULL) AS
   where_clause VARCHAR2(100) := ' WHERE ' || condition;
BEGIN
   IF condition IS NULL THEN where_clause := NULL; END IF;
   EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || where_clause;
EXCEPTION
   ...
END;
```

Using the OPEN-FOR, FETCH, and CLOSE Statements

You use three statements to process a dynamic multi-row query: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set one at a time. When all the rows are processed, you CLOSE the cursor variable. (For more information about cursor variables, see "Using Cursor Variables" on page 5-15.)

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi-row query, executes the query, identifies the result set, positions the cursor on the first row in the result set, then zeroes the rows-processed count kept by %ROWCOUNT.

Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is

```
OPEN {cursor_variable | :host_cursor_variable} FOR dynamic_string
   [USING bind_argument[, bind_argument]...];
```

where cursor_variable is a weakly typed cursor variable (one without a return type), host_cursor_variable is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multi-row query.

In the following example, you declare a cursor variable, then associate it with a dynamic SELECT statement that returns rows from the emp table:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;  -- define weak REF CURSOR type
   emp_cv   EmpCurTyp;  -- declare cursor variable
   my_ename VARCHAR2(15);
   my_sal   NUMBER := 1000;
BEGIN
   OPEN emp_cv FOR  -- open cursor variable
      'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
   ...
END;
```

Any bind arguments in the query are evaluated only when the cursor variable is opened. So, to fetch from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.

Fetching from the Cursor Variable

The FETCH statement returns a row from the result set of a multi-row query, assigns the values of select-list items to corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row. The syntax follows:

```
FETCH {cursor_variable | :host_cursor_variable}
   INTO {define_variable[, define_variable]... | record};
```

Continuing the example, you fetch rows from cursor variable emp_cv into define variables my_ename and my_sal:

```
LOOP
   FETCH emp_cv INTO my_ename, my_sal;  -- fetch next row
   EXIT WHEN emp_cv%NOTFOUND;  -- exit loop when last row is fetched
   -- process row
END LOOP;
```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible variable or field in the INTO clause. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set.

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Closing the Cursor Variable

The CLOSE statement disables a cursor variable. After that, the associated result set is undefined. The syntax follows:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this example, when the last row is processed, you close cursor variable emp_cv:

```
LOOP
   FETCH emp_cv INTO my_ename, my_sal;
   EXIT WHEN emp_cv%NOTFOUND;
   -- process row
END LOOP;
CLOSE emp_cv;  -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises INVALID_CURSOR.

Some Examples

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(100);
   my_job   VARCHAR2(15) := 'CLERK';
BEGIN
   sql_stmt := 'SELECT * FROM emp WHERE job = :j';
   OPEN emp_cv FOR sql_stmt USING my_job;
   LOOP
      FETCH emp_cv INTO emp_rec;
      EXIT WHEN emp_cv%NOTFOUND;
      -- process record
   END LOOP;
   CLOSE emp_cv;
END;
```

The next example illustrates the use of objects and collections. Suppose you define object type `Person` and VARRAY type `Hobbies`, as follows:

```
CREATE TYPE Person AS OBJECT (name VARCHAR2(25), age NUMBER);
CREATE TYPE Hobbies IS VARRAY(10) OF VARCHAR2(25);
```

Now, using dynamic SQL, you can write a package of procedures that uses these types, as follows:

```
CREATE PACKAGE teams AS
   PROCEDURE create_table (tab_name VARCHAR2);
   PROCEDURE insert_row (tab_name VARCHAR2, p Person, h Hobbies);
   PROCEDURE print_table (tab_name VARCHAR2);
END;

CREATE PACKAGE BODY teams AS
   PROCEDURE create_table (tab_name VARCHAR2) IS
   BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE ' || tab_name ||
         ' (p Person, h 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 p, h FROM ' || tab_name;
      LOOP
         FETCH cv INTO p, h;
         EXIT WHEN cv%NOTFOUND;
         -- print attributes of 'p' and elements of 'h'
      END LOOP;
      CLOSE cv;
   END;
END;
```

From an anonymous PL/SQL block, you might call the procedures in package `teams`, as follows:

```
DECLARE
   team_name VARCHAR2(15);
   ...
BEGIN
   ...
   team_name := 'Notables';
   teams.create_table(team_name);
   teams.insert_row(team_name, Person('John', 31),
      Hobbies('skiing', 'coin collecting', 'tennis'));
   teams.insert_row(team_name, Person('Mary', 28),
      Hobbies('golf', 'quilting', 'rock climbing'));
   teams.print_table(team_name);
END;
```

Specifying Parameter Modes

You need not specify a parameter mode for input bind arguments (those used, for example, in the WHERE clause) because the mode defaults to IN. However, you must specify the OUT mode for output bind arguments used in the RETURNING clause of an INSERT, UPDATE, or DELETE statement. An example follows:

```
DECLARE
   sql_stmt VARCHAR2(100);
   old_loc  VARCHAR2(15);
BEGIN
   sql_stmt :=
      'DELETE FROM dept WHERE deptno = 20 RETURNING loc INTO :x';
   EXECUTE IMMEDIATE sql_stmt USING OUT old_loc;
   ...
END;
```

Likewise, when appropriate, you must specify the OUT or IN OUT mode for bind arguments passed as parameters. For example, suppose you want to call the following stand-alone procedure:

```
CREATE PROCEDURE create_dept (
   deptno IN OUT NUMBER,
   dname  IN VARCHAR2,
   loc    IN VARCHAR2) AS
BEGIN
   deptno := deptno_seq.NEXTVAL;
   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(200);
   new_deptno  NUMBER(2);
   new_dname   VARCHAR2(15) := 'ADVERTISING';
   new_loc     VARCHAR2(15) := '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;
```

Tips and Traps

This section shows you how to take full advantage of dynamic SQL and how to avoid some common pitfalls.

Passing the Names of Schema Objects

Suppose you need a procedure that accepts the name of any database table, then drops that table from your schema. Using dynamic SQL, you might write the following stand-alone procedure:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE :tab' USING table_name;
END;
```

However, at run time, this procedure fails with an *invalid table name* error. That is because you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement. Instead, you must embed parameters in the dynamic string, then pass the names of schema objects to those parameters.

To debug the last example, you must revise the `EXECUTE IMMEDIATE` statement. Instead of using a placeholder and bind argument, you use the concatenation operator to embed parameter `table_name` in the dynamic string, as follows:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE ' || table_name;
END;
```

Now, you can pass the name of any database table to the dynamic SQL statement.

Using Duplicate Placeholders

Placeholders in a dynamic SQL statement are associated with bind arguments in the `USING` clause by position, not by name. So, if the same placeholder appears two or more times in the SQL statement, each appearance must correspond to a bind argument in the `USING` clause. For example, given the dynamic string

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

you might code the corresponding `USING` clause as follows:

```
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

However, only the unique placeholders in a dynamic PL/SQL block are associated with bind arguments in the USING clause by position. So, if the same placeholder appears two or more times in a PL/SQL block, all appearances correspond to one bind argument in the USING clause. In the example below, the first unique placeholder (x) is associated with the first bind argument (a). Likewise, the second unique placeholder (y) is associated with the second bind argument (b).

```
DECLARE
   a NUMBER := 4;
   b NUMBER := 7;
BEGIN
   plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;'
   EXECUTE IMMEDIATE plsql_block USING a, b;
   ...
END;
```

Using Cursor Attributes

Every cursor has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor name, they return useful information about the execution of static and dynamic SQL statements.

To process SQL data manipulation statements, Oracle opens an implicit cursor named SQL. Its attributes return information about the most recently executed INSERT, UPDATE, DELETE, or single-row SELECT statement. For example, the following stand-alone function uses %ROWCOUNT to return the number of rows deleted from a database table:

```
CREATE FUNCTION rows_deleted (
   table_name IN VARCHAR2,
   condition IN VARCHAR2) RETURN INTEGER AS
BEGIN
   EXECUTE IMMEDIATE
      'DELETE FROM ' || table_name || ' WHERE ' || condition;
   RETURN SQL%ROWCOUNT;  -- return number of rows deleted
END;
```

Likewise, when appended to a cursor variable name, the cursor attributes return information about the execution of a multi-row query. For more information about cursor attributes, see "Using Cursor Attributes" on page 5-34.

Passing Nulls

Suppose you want to pass nulls to a dynamic SQL statement. For example, you might write the following EXECUTE IMMEDIATE statement:

```
EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING NULL;
```

However, this statement fails with a *bad expression* error because the literal NULL is not allowed in the USING clause. To work around this restriction, simply replace the keyword NULL with an uninitialized variable, as follows:

```
DECLARE
   a_null CHAR(1); -- set to NULL automatically at run time
BEGIN
   EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING a_null;
END;
```

Doing Remote Operations

As the following example shows, PL/SQL subprograms can execute dynamic SQL statements that refer to objects on a remote database:

```
PROCEDURE delete_dept (db_link VARCHAR2, dept_num INTEGER) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM dept@' || db_link ||
      ' WHERE deptno = :n' USING dept_num;
END;
```

Also, the targets of remote procedure calls (RPCs) can contain dynamic SQL statements. For example, suppose the following stand-alone function, which returns the number of rows in a table, resides on the Chicago database:

```
CREATE FUNCTION row_count (tab_name CHAR) RETURN INT AS
   rows INT;
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
   rows INTEGER;
BEGIN
   rows := row_count@chicago('emp');
```

Using Invoker Rights

By default, a stored procedure executes with the privileges of its definer, not its invoker. Such procedures are bound to the schema in which they reside. For example, assume that the following stand-alone procedure, which can drop any kind of database object, resides in schema `scott`:

```
CREATE PROCEDURE drop_it (kind IN VARCHAR2, name IN VARCHAR2) AS
BEGIN
   EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
END;
```

Also assume that user `jones` has been granted the `EXECUTE` privilege on this procedure. When user `jones` calls `drop_it`, as follows, the dynamic `DROP` statement executes with the privileges of user `scott`:

```
SQL> CALL drop_it('TABLE', 'dept');
```

Furthermore, the unqualified reference to table `dept` is resolved in schema `scott`. So, the procedure drops the table from schema `scott`, not from schema `jones`.

However, the `AUTHID` clause enables a stored procedure to execute with the privileges of its invoker (current user). Such procedures are not bound to a particular schema. For example, the following version of `drop_it` executes with the privileges of its invoker:

```
CREATE PROCEDURE drop_it (kind IN VARCHAR2, name IN VARCHAR2)
   AUTHID CURRENT_USER AS
BEGIN
   EXECUTE IMMEDIATE 'DROP ' || kind || ' ' || name;
END;
```

Also, the unqualified reference to the database object is resolved in the schema of the invoker. For details, see "Invoker Rights versus Definer Rights" on page 7-29.

Using Pragma RESTRICT_REFERENCES

A function called from SQL statements must obey certain rules meant to control side effects. (See "Controlling Sides Effects" on page 7-7.) To check for violations of the rules, you can use the pragma `RESTRICT_REFERENCES`, which instructs the compiler to report reads and / or writes to database tables and / or package variables. (See *Oracle8i Application Developer's Guide - Fundamentals*.) However, if the function body contains a dynamic `INSERT`, `UPDATE`, or `DELETE` statement, the function always violates the rules "write no database state" and "read no database state."

Avoiding Deadlocks

In a few situations, executing a SQL data definition statement results in a deadlock. For example, the procedure below causes a deadlock because it attempts to drop itself. To avoid deadlocks, never try to ALTER or DROP a subprogram or package while you are still using it.

```
CREATE PROCEDURE calc_bonus (emp_id NUMBER) AS
BEGIN
   ...
   EXECUTE IMMEDIATE 'DROP PROCEDURE calc_bonus';
   -- causes "timeout occurred while waiting to lock object" error
END;
```

11

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 sequenced to form PL/SQL statements. Also, to save you time and trouble, it provides usage notes and short examples.

Major Topics

Reading the Syntax Diagrams

When you are unsure of the syntax to use in a PL/SQL statement, trace through its syntax diagram, reading from left to right and top to bottom. You can verify or construct any PL/SQL statement that way.

The diagrams are graphic representations of Bachus-Naur Form (BNF) productions. Within the diagrams, keywords are enclosed in boxes, delimiters in circles, and identifiers in ovals.

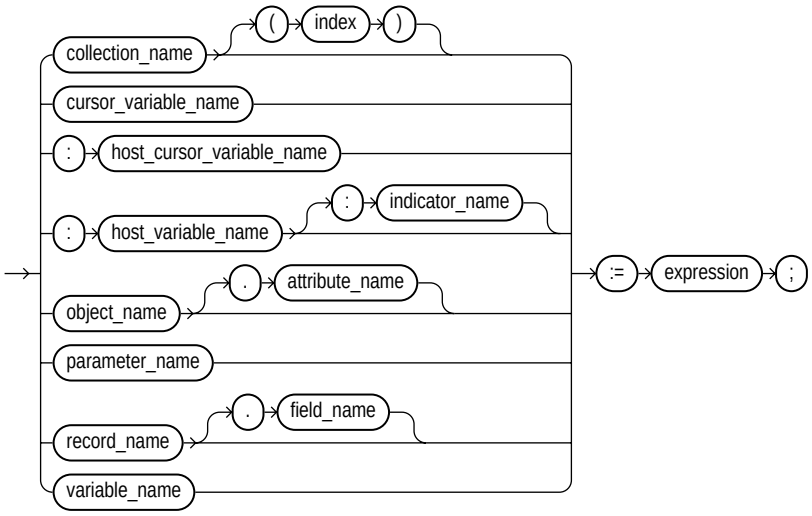
Each diagram defines a syntactic element. Every path through the diagram describes a possible form of that element. Follow in the direction of the arrows. If a line loops back on itself, you can repeat the element enclosed by the loop.

Assignment Statement

An assignment statement sets the current value of a variable, field, parameter, or element. The statement consists of an assignment target followed by the assignment operator and an expression. When the statement is executed, the expression is evaluated and the resulting value is stored in the target. For more information, see "Assignments" on page 2-40.

Syntax

assignment_statement



Keyword and Parameter Description

collection_name

This identifies a nested table, index-by table, or varray previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

host_cursor_variable_name

This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

host_variable_name

This identifies a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Host variables must be prefixed with a colon.

object_name

This identifies an object (instance of an object type) previously declared within the current scope.

indicator_name

This identifies an indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable "indicates" the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables let you detect nulls or truncated values in output host variables.

parameter_name

This identifies a formal `OUT` or `IN OUT` parameter of the subprogram in which the assignment statement appears.

index

This is a numeric expression that must yield a value of type `BINARY_INTEGER` or a value implicitly convertible to that datatype.

record_name.field_name

This identifies a field in a user-defined or `%ROWTYPE` record previously declared within the current scope.

variable_name

This identifies a PL/SQL variable previously declared within the current scope.

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of `expression`, see "Expressions" on page 11-63. 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.

Usage Notes

By default, unless a variable is initialized in its declaration, it is initialized to `NULL` every time a block or subprogram is entered. So, never reference a variable before you assign it a value.

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

Only the values `TRUE`, `FALSE`, and `NULL` can be assigned to a Boolean variable. When applied to an expression, the relational operators return a Boolean value. So, the following assignment is legal:

```
DECLARE
   out_of_range BOOLEAN;
   ...
BEGIN
   ...
   out_of_range := (salary < minimum) OR (salary > maximum);
```

As the next example shows, you can assign the value of an expression to a specific field in a record:

```
DECLARE
   emp_rec emp%ROWTYPE;
BEGIN
   ...
   emp_rec.sal := current_salary + increase;
```

Moreover, you can assign values to all fields in a record at once. PL/SQL allows aggregate assignment between entire records if their declarations refer to the same cursor or table. For example, the following assignment is legal:

```
DECLARE
   emp_rec1 emp%ROWTYPE;
   emp_rec2 emp%ROWTYPE;
   dept_rec dept%ROWTYPE;
BEGIN
   ...
   emp_rec1 := emp_rec2;
```

The next assignment is illegal because you cannot use the assignment operator to assign a list of values to a record:

```
dept_rec := (60, 'PUBLICITY', 'LOS ANGELES');
```

Using the following syntax, you can assign the value of an expression to a specific element in a collection:

```
collection_name(index) := expression;
```

In the following example, you assign the uppercase value of `last_name` to the third row in nested table `ename_tab`:

```
ename_tab(3) := UPPER(last_name);
```

Examples

Several examples of assignment statements follow:

```
wages := hours_worked * hourly_salary;
country := 'France';
costs := labor + supplies;
done := (count > 100);
dept_rec.loc := 'BOSTON';
comm_tab(5) := sales * 0.15;
```

Related Topics

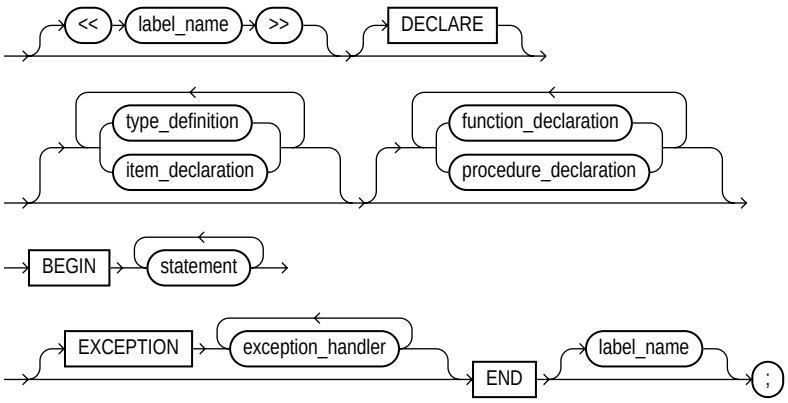
Constants and Variables, Expressions, SELECT INTO Statement

Blocks

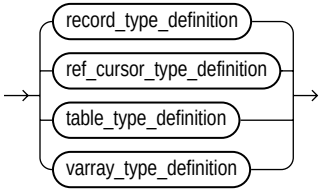
The basic program unit in PL/SQL is the block. A PL/SQL block is defined by the keywords `DECLARE`, `BEGIN`, `EXCEPTION`, and `END`. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required. You can nest a block within another block wherever you can place an executable statement. For more information, see "Block Structure" on page 1-2 and "Scope and Visibility" on page 2-37.

Syntax

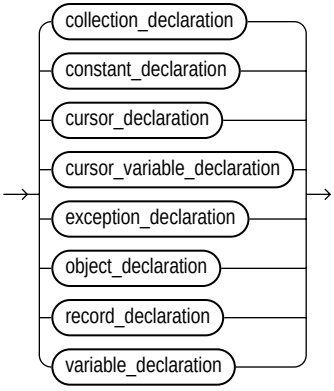
plsql_block



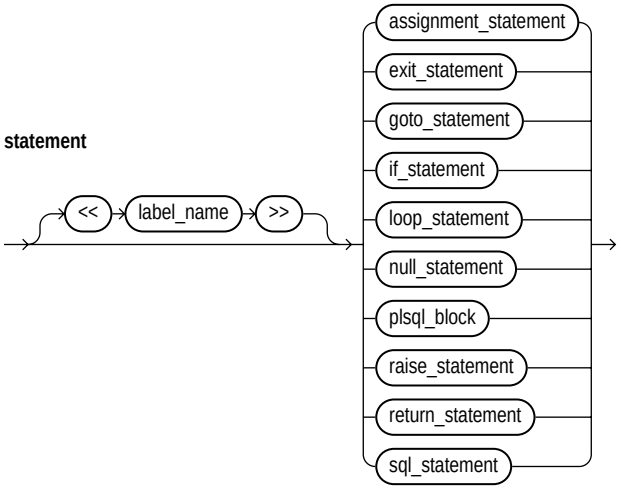
type_definition



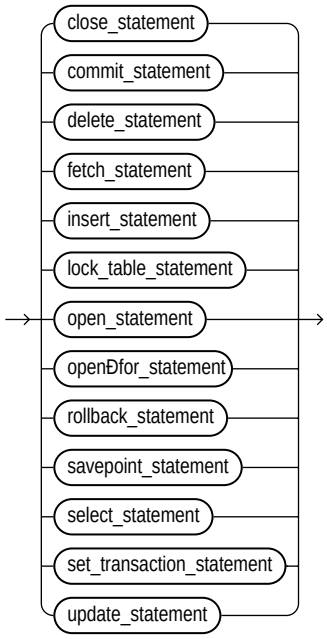
item_declaration



statement



sql_statement



Keyword and Parameter Description

label_name

This is an undeclared identifier that optionally labels a PL/SQL block. If used, `label_name` must be enclosed by double angle brackets and must appear at the beginning of the block. Optionally, `label_name` (*not* enclosed by angle brackets) can also appear at the end of the block.

A global identifier declared in an enclosing block can be redeclared in a sub-block, in which case the local declaration prevails and the sub-block cannot reference the global identifier. To reference the global identifier, you must 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
         ...
      END IF;
   END;
END outer;
```

DECLARE

This keyword signals the start of the declarative part of a PL/SQL block, which contains local declarations. Items declared locally exist only within the current block and all its sub-blocks and are not visible to enclosing blocks. The declarative part of a PL/SQL block is optional. It is terminated implicitly by the keyword `BEGIN`, which introduces the executable part of the block.

PL/SQL does not allow forward references. So, you must declare an item before referencing it in other statements, including other declarative statements. Also, you must declare subprograms at the end of a declarative section after all other program items.

collection_declaration

This identifies an index-by table, nested table, or varray previously declared within the current scope. For the syntax of `collection_declaration`, see "Collections" on page 11-21.

constant_declaration

This construct declares a constant. For the syntax of `constant_declaration`, see "Constants and Variables" on page 11-30.

cursor_declaration

This construct declares an explicit cursor. For the syntax of `cursor_declaration`, see "Cursors" on page 11-45.

cursor_variable_declaration

This construct declares a cursor variable. For the syntax of `cursor_variable_declaration`, see "Cursor Variables" on page 11-39.

exception_declaration

This construct declares an exception. For the syntax of `exception_declaration`, see "Exceptions" on page 11-55.

object_declaration

This identifies an object (instance of an object type) previously declared within the current scope. For the syntax of `object_declaration`, see "Object Types" on page 11-105.

record_declaration

This construct declares a user-defined record. For the syntax of `record_declaration`, see "Records" on page 11-134.

variable_declaration

This construct declares a variable. For the syntax of `variable_declaration`, see "Constants and Variables" on page 11-30.

function_declaration

This construct declares a function. For the syntax of `function_declaration`, see "Functions" on page 11-79.

procedure_declaration

This construct declares a procedure. For the syntax of `procedure_declaration`, see "Procedures" on page 11-127.

BEGIN

This keyword signals the start of the executable part of a PL/SQL block, which contains executable statements. The executable part of a block is required. That is, a PL/SQL block must contain at least one executable statement. The `NULL` statement meets this requirement.

statement

This is an executable (not declarative) statement that you use to create algorithms. A sequence of statements can include procedural statements such as `RAISE`, SQL statements such as `UPDATE`, and PL/SQL blocks (sometimes called "block statements").

PL/SQL statements are free format. That is, they can continue from line to line if you do not split keywords, delimiters, or literals across lines. A semicolon (;) serves as the statement terminator.

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.

EXCEPTION

This keyword signals the start of the exception-handling part of a PL/SQL block. When an exception is raised, normal execution of the block stops and control transfers to the appropriate exception handler. After the exception handler completes, execution proceeds with the statement following the block.

If there is no exception handler for the raised exception in the current block, control passes to the enclosing block. This process repeats until an exception handler is found or there are no more enclosing blocks. If PL/SQL can find no exception handler for the exception, execution stops and an *unhandled exception* error is returned to the host environment. For more information, see Chapter 6.

exception_handler

This construct 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 11-55.

END

This keyword signals the end of a PL/SQL block. It must be the last keyword in a block. Neither the END IF in an IF statement nor the END LOOP in a LOOP statement can substitute for the keyword END.

END does *not* signal the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.

Example

The following PL/SQL block declares several variables and constants, then calculates a ratio using values selected from a database table:

```
-- available online in file 'examp11'
DECLARE
   numerator   NUMBER;
   denominator NUMBER;
   the_ratio   NUMBER;
```

```
   lower_limit CONSTANT NUMBER := 0.72;
   samp_num    CONSTANT NUMBER := 132;
BEGIN
   SELECT x, y INTO numerator, denominator FROM result_table
      WHERE sample_id = samp_num;
   the_ratio := numerator/denominator;
   IF the_ratio > lower_limit THEN
      INSERT INTO ratio VALUES (samp_num, the_ratio);
   ELSE
      INSERT INTO ratio VALUES (samp_num, -1);
   END IF;
   COMMIT;
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      INSERT INTO ratio VALUES (samp_num, 0);
      COMMIT;
   WHEN OTHERS THEN
      ROLLBACK;
END;
```

Related Topics

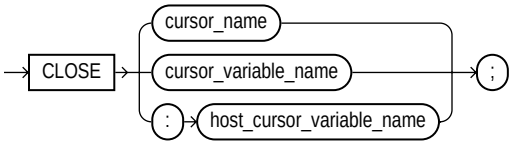
Constants and Variables, Exceptions, Functions, Procedures

CLOSE Statement

The CLOSE statement allows resources held by an open cursor or cursor variable to be reused. No more rows can be fetched from a closed cursor or cursor variable. For more information, see "Managing Cursors" on page 5-6.

Syntax

close_statement



Keyword and Parameter Description

cursor_name

This identifies an explicit cursor previously declared within the current scope and currently open.

cursor_variable_name

This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope and currently open.

host_cursor_variable_name

This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

Usage Notes

Once a cursor or cursor variable is closed, you can reopen it using the OPEN or OPEN-FOR statement, respectively. If you reopen a cursor without closing it first, PL/SQL raises the predefined exception CURSOR_ALREADY_OPEN. However, you need not close a cursor variable before reopening it.

If you try to close an already-closed or never-opened cursor or cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Example

In the following example, after the last row is fetched and processed, you close the cursor variable emp_cv:

```
LOOP
   FETCH emp_cv INTO emp_rec;
   EXIT WHEN emp_cv%NOTFOUND;
   ...  -- process data record
END LOOP;
/* Close cursor variable. */
CLOSE emp_cv;
```

Related Topics

FETCH Statement, OPEN Statement, OPEN-FOR Statement

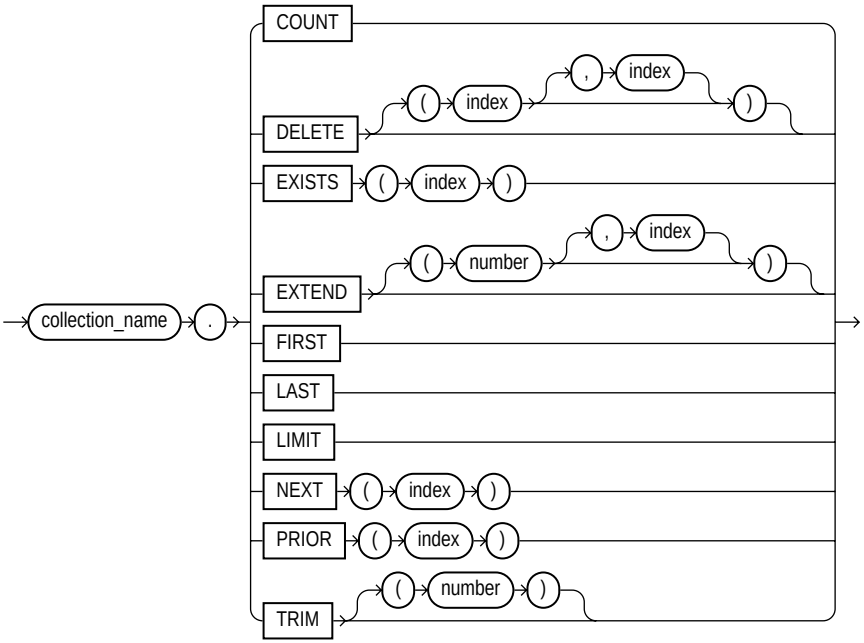
Collection Methods

A collection method is a built-in function or procedure that operates on collections and is called using dot notation. The methods EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, NEXT, EXTEND, TRIM, and DELETE help generalize code, make collections easier to use, and make your applications easier to maintain.

EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, and NEXT are functions, which appear as part of an expression. EXTEND, TRIM, and DELETE are procedures, which appear as a statement. EXISTS, PRIOR, NEXT, TRIM, EXTEND, and DELETE take integer parameters. For more information, see "Using Collection Methods" on page 4-21.

Syntax

collection_method_call



Keyword and Parameter Description

collection_name

This identifies an index-by table, nested table, or varray previously declared within the current scope.

COUNT

COUNT returns the number of elements that a collection currently contains, which is useful because the current size of a collection is not always known. You can use COUNT wherever an integer expression is allowed.

For varrays, COUNT always equals LAST. For nested tables, normally, COUNT equals LAST. But, if you delete elements from the middle of a nested table, COUNT is smaller than LAST.

DELETE

This procedure has three forms. DELETE removes all elements from a collection. DELETE(n) removes the *n*th element from a nested table. If n is null, DELETE(n) does nothing. DELETE(m,n) removes all elements in the range m..n from a nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing.

index

This is an expression that must yield (or convert implicitly to) an integer. For more information, see "Datatype Conversion" on page 2-27.

EXISTS

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

EXTEND

This procedure has three forms. EXTEND appends one null element to a collection. EXTEND(n) appends n null elements to a collection. EXTEND(n,i) appends n copies of the *i*th element to a collection. EXTEND operates on the internal size of a collection. So, if EXTEND encounters deleted elements, it includes them in its tally.

FIRST, LAST

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection. If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same index number.

For varrays, FIRST always returns 1 and LAST always equals COUNT. For nested tables, normally, LAST equals COUNT. But, if you delete elements from the middle of a nested table, LAST is larger than COUNT.

LIMIT

For nested tables, which have no maximum size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain (which you must specify in its type definition).

NEXT, PRIOR

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

TRIM

This procedure has two forms. TRIM removes one element from the end of a collection. TRIM(n) removes n elements from the end of a collection. If n is greater than COUNT, TRIM(n) raises SUBSCRIPT_BEYOND_COUNT.

TRIM operates on the internal size of a collection. So, if TRIM encounters deleted elements, it includes them in its tally.

Usage Notes

You cannot use collection methods in a SQL statement. If you try, you get a compilation error.

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

You can use PRIOR or NEXT to traverse collections indexed by any series of subscripts. For example, you can use PRIOR or NEXT to traverse a nested table from which some elements have been deleted.

EXTEND operates on the internal size of a collection, which includes deleted elements. You cannot use EXTEND to initialize an atomically null collection. Also, if you impose the NOT NULL constraint on a TABLE or VARRAY type, you cannot apply the first two forms of EXTEND to collections of that type.

If an element to be deleted does not exist, DELETE simply skips it; no exception is raised. Varrays are dense, so you cannot delete their individual elements.

PL/SQL keeps placeholders for deleted elements. So, you can replace a deleted element simply by assigning it a new value. However, PL/SQL does not keep placeholders for trimmed elements.

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

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

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. So, you can apply methods FIRST, LAST, COUNT, and so on to such parameters. For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

Examples

In the following example, you use NEXT to traverse a nested table from which some elements have been deleted:

```
i := courses.FIRST;  -- get subscript of first element
WHILE i IS NOT NULL LOOP
   -- do something with courses(i)
   i := courses.NEXT(i);  -- get subscript of next element
END LOOP;
```

In the following example, PL/SQL executes the assignment statement only if element i exists:

```
IF courses.EXISTS(i) THEN
   courses(i) := new_course;
END IF;
```

The next example shows that you can use FIRST and LAST to specify the lower and upper bounds of a loop range provided each element in that range exists:

```
FOR i IN courses.FIRST..courses.LAST LOOP ...
```

In the following example, you delete elements 2 through 5 from a nested table:

```
courses.DELETE(2, 5);
```

In the final example, you use `LIMIT` to determine if you can add 20 more elements to varray `projects`:

```
IF (projects.COUNT + 20) < projects.LIMIT THEN
   -- add 20 more elements
```

Related Topics

Collections

Collections

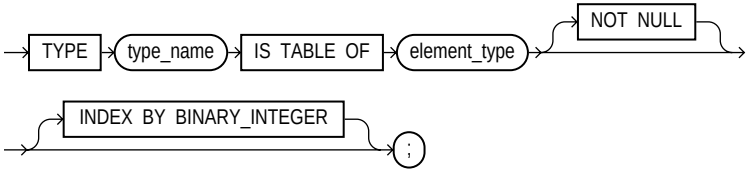
A collection is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers three kinds of collections: index-by tables, nested tables, and varrays (short for variable-size arrays). Nested tables extend the functionality of index-by tables (formerly called "PL/SQL tables").

Collections work like the arrays found in most third-generation programming languages. However, collections can have only one dimension and must be indexed by integers. (In some languages such as Ada and Pascal, arrays can have multiple dimensions and can be indexed by enumeration types.)

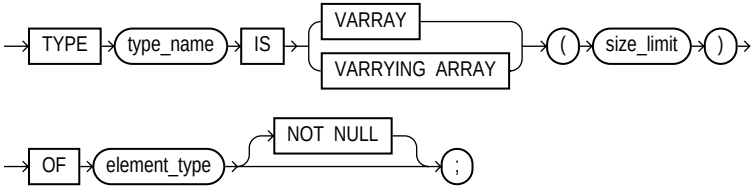
Nested tables and varrays can store instances of an object type and, conversely, can be attributes of an object type. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. For more information, see "Defining and Declaring Collections" on page 4-5.

Syntax

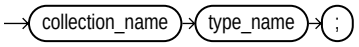
table_type_definition



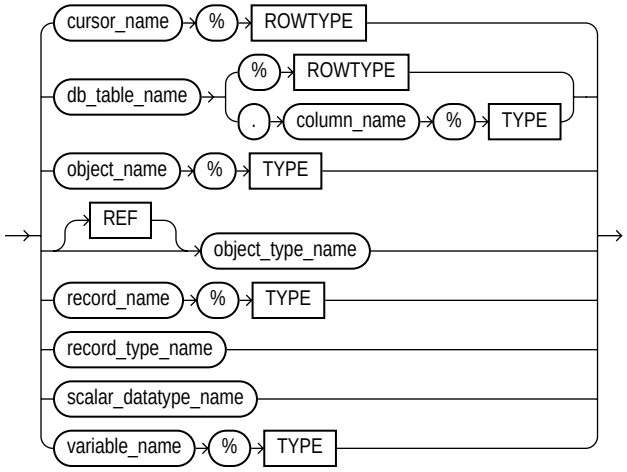
varray_type_definition



collection_declaration



element_type



Keyword and Parameter Description

type_name

This identifies a user-defined type specifier, which is used in subsequent declarations of collections.

element_type

This is any PL/SQL datatype except `BINARY_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `NCHAR`, `NCLOB`, `NVARCHAR2`, object types with `TABLE` or `VARRAY` attributes, `PLS_INTEGER`, `POSITIVE`, `POSITIVEN`, `REF CURSOR`, `SIGNTYPE`, `STRING`, `TABLE`, or `VARRAY`. Also, with varrays, `element_type` cannot be `BLOB`, `CLOB`, or an object type with `BLOB` or `CLOB` attributes. If `element_type` is a record type, every field in the record must be a scalar type or an object type.

INDEX BY BINARY_INTEGER

This optional clause lets you define Version 2 PL/SQL tables, which are called index-by tables in Version 8.

size_limit

This is a positive integer literal that specifies the maximum size of a varray, which is the maximum number of elements the varray can contain.

Usage Notes

Nested tables extend the functionality of index-by tables, so they differ in several ways. See "Nested Tables versus Index-by Tables" on page 4-3.

Every element reference includes the collection name and a subscript enclosed in parentheses; the subscript determines which element is processed. Except for index-by tables, which can have negative subscripts, collection subscripts have a fixed lower bound of 1.

You can define all three collection types in the declarative part of any PL/SQL block, subprogram, or package. But, only nested table and varray types can be `CREATE`d and stored in an Oracle database.

Index-by tables and nested tables can be sparse (have non-consecutive subscripts), but varrays are always dense (have consecutive subscripts). Unlike nested tables, varrays retain their ordering and subscripts when stored in the database.

Initially, index-by tables are sparse. That enables you, for example, to store reference data in a temporary index-by table using a numeric primary key (account numbers or employee numbers for example) as the index.

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

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

Because nested tables and varrays can be atomically null, they can be tested for nullity. However, they cannot be compared for equality or inequality. This restriction also applies to implicit comparisons. For example, collections cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` list.

Collections can store instances of an object type and, conversely, can be attributes of an object type. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

When calling a function that returns a collection, you use the following syntax to reference elements in the collection:

```
collection_name(parameter_list)(subscript)
```

With the Oracle Call Interface (OCI) or the Oracle Precompilers, you can bind host arrays to collections declared as the formal parameters of a subprogram. That allows you to pass host arrays to stored functions and procedures.

Examples

To specify the element type of a collection, you can use %TYPE or %ROWTYPE, as the following example shows:

```
DECLARE
   TYPE JobList IS VARRAY(10) OF emp.job%TYPE;  -- based on column
   CURSOR c1 IS SELECT * FROM dept;
   TYPE DeptFile IS TABLE OF c1%ROWTYPE;  -- based on cursor
   TYPE EmpFile IS VARRAY(150) OF emp%ROWTYPE;  -- based on database
table
```

In the next example, you use a RECORD type to specify the element type:

```
DECLARE
   TYPE Entry IS RECORD (
      term    VARCHAR2(20),
      meaning VARCHAR2(200));
   TYPE Glossary IS VARRAY(250) OF Entry;
```

In the example below, you declare an index-by table of records. Each element of the table stores a row from the emp database table.

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab EmpTabTyp;
BEGIN
   /* Retrieve employee record. */
   SELECT * INTO emp_tab(7468) FROM emp WHERE empno = 7788;
```

When defining a VARRAY type, you must specify its maximum size. In the following example, you define a type that stores up to 366 dates:

```
DECLARE
   TYPE Calendar IS VARRAY(366) OF DATE;
```

Once you define a collection type, you can declare collections of that type, as the following SQL*Plus script shows:

```
CREATE TYPE Project AS OBJECT(
   project_no NUMBER(2),
   title      VARCHAR2(35),
   cost       NUMBER(7,2))
/
CREATE TYPE ProjectList AS VARRAY(50) OF Project  -- VARRAY type
/
CREATE TABLE department (
   idnum    NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
   projects ProjectList)  -- declare varray
/
```

The identifier `projects` represents an entire varray. Each element of `projects` will store a `Project` object.

In the following example, you declare a nested table as the formal parameter of a packaged procedure:

```
CREATE PACKAGE personnel AS
   TYPE Staff IS TABLE OF Employee;
   ...
   PROCEDURE award_bonuses (members IN Staff);
```

You can specify a collection type in the RETURN clause of a function spec, as the following example shows:

```
DECLARE
   TYPE SalesForce IS VARRAY(20) OF Salesperson;
   FUNCTION top_performers (n INTEGER) RETURN SalesForce IS ...
```

In the following example, you update the list of projects assigned to the Security Department:

```
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';
```

In the next example, you retrieve all the projects for the Accounting Department into a local varray:

```
DECLARE
   my_projects ProjectList;
BEGIN
   SELECT projects INTO my_projects FROM department
      WHERE name = 'Accounting';
```

Related Topics

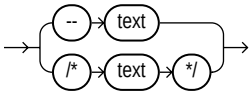
Collection Methods, Object Types, Records

Comments

Comments describe the purpose and use of code segments and so promote readability. PL/SQL supports two comment styles: single-line and multi-line. Single-line comments begin with a double hyphen (`--`) anywhere on a line and extend to the end of the line. Multi-line comments begin with a slash-asterisk (`/*`), end with an asterisk-slash (`*/`), and can span multiple lines. For more information, see "Comments" on page 2-9.

Syntax

comment



Usage Notes

Comments can appear within a statement at the end of a line. However, you cannot nest comments.

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

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

```
-- UPDATE dept SET loc = my_loc WHERE deptno = my_deptno;
```

You can use multi-line comment delimiters to comment-out whole sections of code.

Examples

The following examples show various comment styles:

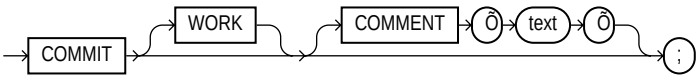
```
-- compute the area of a circle
area := pi * radius**2;  -- pi equals 3.14159
/* Compute the area
   of a circle. */
area := pi * radius**2;  /* pi equals 3.14159 */
```

COMMIT Statement

The COMMIT statement explicitly makes permanent any changes made to the database during the current transaction. Changes made to the database are not considered permanent until they are committed. A commit also makes the changes visible to other users. For more information, see "Processing Transactions" on page 5-40.

Syntax

commit_statement



Keyword and Parameter Description

WORK

This keyword is optional and has no effect except to improve readability.

COMMENT

This keyword specifies a comment to be associated with the current transaction and is typically used with distributed transactions. The text must be a quoted literal no more than 50 characters long.

Usage Notes

The COMMIT statement releases all row and table locks. It also erases any savepoints you marked since the last commit or rollback. Until your changes are committed, the following conditions hold:

- You can see the changes when you query the tables you modified, but other users cannot see the changes.
- If you change your mind or need to correct a mistake, you can use the ROLLBACK statement to roll back (undo) the changes.

If you commit while a FOR UPDATE cursor is open, a subsequent fetch on that cursor raises an exception. The cursor remains open, however, so you should close it. For more information, see "Using FOR UPDATE" on page 5-47.

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 *Oracle8i Concepts*.

In SQL, the FORCE clause manually commits an in-doubt distributed transaction. However, PL/SQL does not support this clause. For example, the following statement is illegal:

```
COMMIT WORK FORCE '23.51.54';  -- illegal
```

In embedded SQL, the RELEASE option frees all Oracle resources (locks and cursors) held by a program and disconnects from the database. However, PL/SQL does not support this option. For example, the following statement is illegal:

```
COMMIT WORK RELEASE;  -- illegal
```

Related Topics

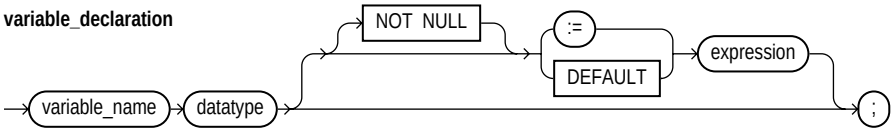
ROLLBACK Statement, SAVEPOINT Statement

Constants and Variables

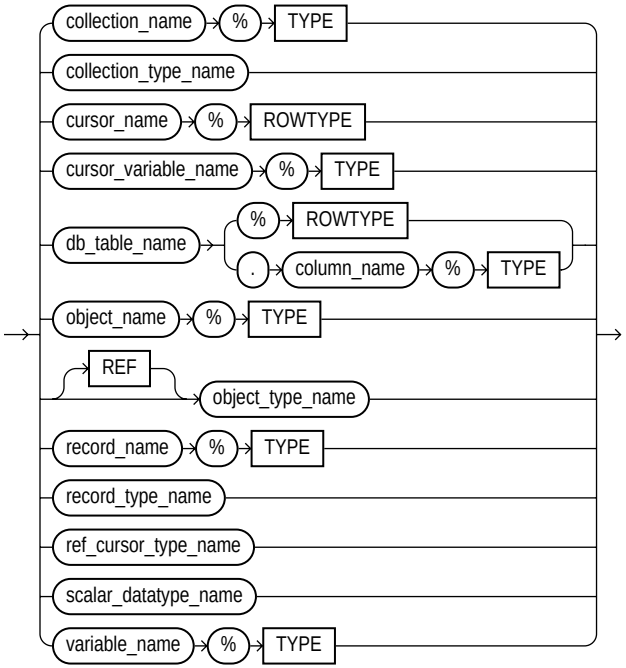
You can declare constants and variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the NOT NULL constraint. For more information, see "Declarations" on page 2-29.

Syntax

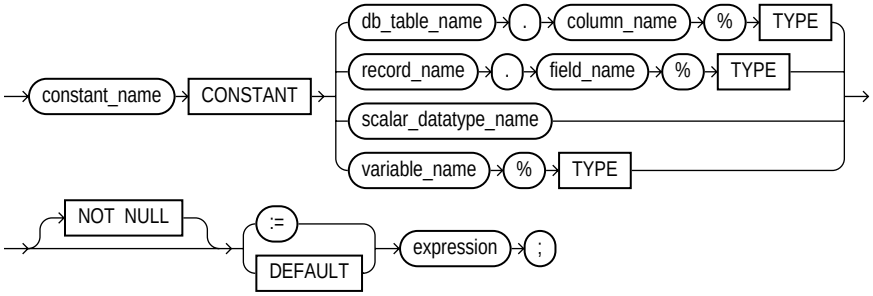
variable_declaration



datatype



constant_declaration



Keyword and Parameter Description

constant_name

This identifies a program constant. For naming conventions, see "Identifiers" on page 2-4.

CONSTANT

This keyword denotes the declaration of a constant. You must initialize a constant in its declaration. Once initialized, the value of a constant cannot be changed.

record_name.field_name

This identifies a field in a user-defined or %ROWTYPE record previously declared within the current scope.

scalar_type_name

This identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2. For more information, see "Datatypes" on page 2-10.

db_table_name.column_name

This identifies a database table and column that must be accessible when the declaration is elaborated.

variable_name

This identifies a program variable.

collection_name

This identifies a nested table, index-by table, or varray previously declared within the current scope.

cursor_name

This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable previously declared within the current scope.

object_name

This identifies an object (instance of an object type) previously declared within the current scope.

record_name

This identifies a user-defined record previously declared within the current scope.

db_table_name

This identifies a database table (or view) that must be accessible when the declaration is elaborated.

%ROWTYPE

This attribute provides a record type that represents a row in a database table or a row fetched from a previously declared cursor. Fields in the record and corresponding columns in the row have the same names and datatypes.

%TYPE

This attribute provides the datatype of a previously declared collection, cursor variable, field, object, record, database column, or variable.

NOT NULL

This constraint prevents the assigning of nulls to a variable or constant. At run time, trying to assign a null to a variable defined as `NOT NULL` raises the predefined exception `VALUE_ERROR`. The constraint `NOT NULL` must be followed by an initialization clause.

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the constant or variable. The value and the constant or variable must have compatible datatypes.

Usage Notes

Constants and variables are initialized every time a block or subprogram is entered. By default, variables are initialized to `NULL`. So, unless you expressly initialize a variable, its value is undefined.

Whether public or private, constants and variables declared in a package spec are initialized only once per session.

An initialization clause is required when declaring `NOT NULL` variables and when declaring constants.

You cannot use the attribute `%ROWTYPE` to declare a constant. If you use `%ROWTYPE` to declare a variable, initialization is not allowed.

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;
last_name    VARCHAR2(20);
my_ename     emp.ename%TYPE;
```

Related Topics

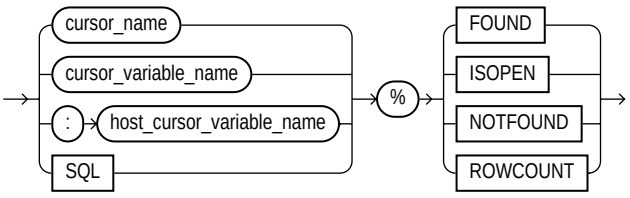
Assignment Statement, Expressions, %ROWTYPE Attribute, %TYPE Attribute

Cursor Attributes

Cursors and cursor variables have four attributes that give you useful information about the execution of a data manipulation statement. For more information, see "Using Cursor Attributes" on page 5-34.

Syntax

cursor_attribute



Keyword and Parameter Description

cursor_name

This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope.

host_cursor_variable_name

This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

SQL

This is the name of the implicit SQL cursor. For more information, see "SQL Cursor" on page 11-151.

%FOUND

This is a cursor attribute, which can be appended to the name of a cursor or cursor variable. Before the first fetch from an open cursor, cursor_name%FOUND yields NULL. Thereafter, it yields TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row.

Until a SQL statement is executed, SQL%FOUND yields NULL. Thereafter, it yields TRUE if the statement affected any rows, or FALSE if it affected no rows.

%ISOPEN

This is a cursor attribute, which can be appended to the name of a cursor or cursor variable. If a cursor is open, cursor_name%ISOPEN yields TRUE; otherwise, it yields FALSE.

Oracle automatically closes the implicit SQL cursor after executing its associated SQL statement, so SQL%ISOPEN always yields FALSE.

%NOTFOUND

This is a cursor attribute, which can be appended to the name of a cursor or cursor variable. Before the first fetch from an open cursor, cursor_name%NOTFOUND yields NULL. Thereafter, it yields FALSE if the last fetch returned a row, or TRUE if the last fetch failed to return a row.

Until a SQL statement is executed, SQL%NOTFOUND yields NULL. Thereafter, it yields FALSE if the statement affected any rows, or TRUE if it affected no rows.

%ROWCOUNT

This is a cursor attribute, which can be appended to the name of a cursor or cursor variable. When a cursor is opened, %ROWCOUNT is zeroed. Before the first fetch, cursor_name%ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the latest fetch returned a row.

Until a SQL statement is executed, SQL%ROWCOUNT yields NULL. Thereafter, it yields the number of rows affected by the statement. SQL%ROWCOUNT yields 0 if the statement affected no rows.

Usage Notes

The cursor attributes apply to every cursor or cursor variable. So, for example, you can open multiple cursors, then use %FOUND or %NOTFOUND to tell which cursors have rows left to fetch. Likewise, you can use %ROWCOUNT to tell how many rows have been fetched so far.

If a cursor or cursor variable is not open, referencing it with %FOUND, %NOTFOUND, or %ROWCOUNT raises the predefined exception INVALID_CURSOR.

When a cursor or cursor variable is opened, the rows that satisfy the associated query are identified and form the result set. Rows are fetched from the result set one at a time.

If a SELECT INTO statement returns more than one row, PL/SQL raises the predefined exception TOO_MANY_ROWS and sets %ROWCOUNT to 1, not the actual number of rows that satisfy the query.

Before the first fetch, %NOTFOUND evaluates to NULL. So, if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement instead:

```
EXIT WHEN c1%NOTFOUND OR ci%NOTFOUND IS NULL;
```

You can use the cursor attributes in procedural statements but *not* in SQL statements.

Examples

The PL/SQL block below uses %FOUND to select an action. The IF statement either inserts a row or exits the loop unconditionally.

```
-- available online in file 'examp12'
DECLARE
   CURSOR num1_cur IS SELECT num FROM num1_tab
      ORDER BY sequence;
   CURSOR num2_cur IS SELECT num FROM num2_tab
      ORDER BY sequence;
   num1     num1_tab.num%TYPE;
   num2     num2_tab.num%TYPE;
   pair_num NUMBER := 0;
BEGIN
   OPEN num1_cur;
   OPEN num2_cur;
   LOOP   -- loop through the two tables and get pairs of numbers
      FETCH num1_cur INTO num1;
      FETCH num2_cur INTO num2;
      IF (num1_cur%FOUND) AND (num2_cur%FOUND) THEN
         pair_num := pair_num + 1;
         INSERT INTO sum_tab VALUES (pair_num, num1 + num2);
```

```
      ELSE
         EXIT;
      END IF;
   END LOOP;
   CLOSE num1_cur;
   CLOSE num2_cur;
END;
```

The next example uses the same block. However, instead of using %FOUND in an IF statement, it uses %NOTFOUND in an EXIT WHEN statement.

```
-- available online in file 'examp13'
DECLARE
   CURSOR num1_cur IS SELECT num FROM num1_tab
      ORDER BY sequence;
   CURSOR num2_cur IS SELECT num FROM num2_tab
      ORDER BY sequence;
   num1     num1_tab.num%TYPE;
   num2     num2_tab.num%TYPE;
   pair_num NUMBER := 0;
BEGIN
   OPEN num1_cur;
   OPEN num2_cur;
   LOOP   -- loop through the two tables and get
          -- pairs of numbers
      FETCH num1_cur INTO num1;
      FETCH num2_cur INTO num2;
      EXIT WHEN (num1_cur%NOTFOUND) OR (num2_cur%NOTFOUND);
      pair_num := pair_num + 1;
      INSERT INTO sum_tab VALUES (pair_num, num1 + num2);
   END LOOP;
   CLOSE num1_cur;
   CLOSE num2_cur;
END;
```

In the following example, you use %ISOPEN to make a decision:

```
IF NOT (emp_cur%ISOPEN) THEN
   OPEN emp_cur;
END IF;
FETCH emp_cur INTO emp_rec;
```

The following PL/SQL block uses %ROWCOUNT to fetch the names and salaries of the five highest-paid employees:

```
-- available online in file 'examp14'
DECLARE
   CURSOR c1 is
   SELECT ename, empno, sal FROM emp
      ORDER BY sal DESC;   -- start with highest-paid employee
   my_ename CHAR(10);
   my_empno NUMBER(4);
   my_sal   NUMBER(7,2);
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_ename, my_empno, my_sal;
      EXIT WHEN (c1%ROWCOUNT > 5) OR (c1%NOTFOUND);
      INSERT INTO temp VALUES (my_sal, my_empno, my_ename);
      COMMIT;
   END LOOP;
   CLOSE c1;
END;
```

In the final example, you use %ROWCOUNT to raise an exception if an unexpectedly high number of rows is deleted:

```
DELETE FROM accts WHERE status = 'BAD DEBT';
IF SQL%ROWCOUNT > 10 THEN
     RAISE out_of_bounds;
END IF;
```

Related Topics

Cursors, Cursor Variables

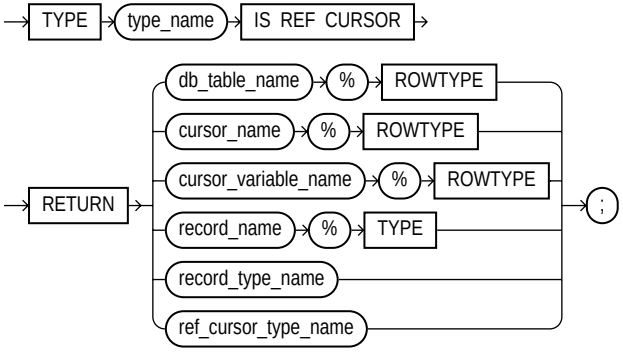
Cursor Variables

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use an explicit cursor, which names the work area. Or, you can use a cursor variable, which points to the work area. Whereas a cursor always refers to the same query work area, a cursor variable can refer to different work areas. To create cursor variables, you define a REF CURSOR type, then declare cursor variables of that type.

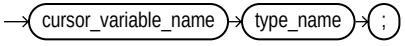
Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, *not* an item. For more information, see "Using Cursor Variables" on page 5-15.

Syntax

ref_cursor_type_definition



cursor_variable_declaration



Keyword and Parameter Description

type_name

This is a user-defined type specifier, which is used in subsequent declarations of PL/SQL cursor variables.

REF CURSOR

In PL/SQL, pointers have datatype REF X, where REF is short for REFERENCE and X stands for a class of objects. Therefore, cursor variables have datatype REF CURSOR.

RETURN

This keyword introduces the RETURN clause, which specifies the datatype of a cursor variable result value. You can use the %ROWTYPE attribute in the RETURN clause to provide a record type that represents a row in a database table or a row returned by a cursor or strongly typed cursor variable. Also, you can use the %TYPE attribute to provide the datatype of a previously declared record.

cursor_name

This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable previously declared within the current scope.

record_name

This identifies a user-defined record previously declared within the current scope.

record_type_name

This identifies a RECORD type previously defined within the current scope.

db_table_name

This identifies a database table (or view) that must be accessible when the declaration is elaborated.

%ROWTYPE

This attribute provides a record type that represents a row in a database table or a row fetched from a cursor or strongly typed cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.

%TYPE

This attribute provides the datatype of a previously declared user-defined record.

Usage Notes

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as a bind variable to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side.

The Oracle database server also has a PL/SQL engine. So, you can pass cursor variables back and forth between an application and server via remote procedure calls (RPCs). And, if you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side.

Mainly, you use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle server can all refer to the same work area.

REF CURSOR types can be *strong* (restrictive) or *weak* (nonrestrictive). A strong REF CURSOR type definition specifies a return type, but a weak definition does not. Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Once you define a REF CURSOR type, you can declare cursor variables of that type. Yu 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.

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 one at a time. 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. For more information, see "Using Cursor Attributes" on page 5-34.

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.

Currently, cursor variables are subject to several restrictions. See "Restrictions on Cursor Variables" on page 5-33.

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 emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                          choice IN NUMBER);
END emp_data;

CREATE PACKAGE BODY emp_data AS
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                          choice IN NUMBER) IS
   BEGIN
      IF choice = 1 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
      ELSIF choice = 2 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
      ELSIF choice = 3 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
      END IF;
   END open_emp_cv;
END emp_data;
```

Alternatively, you can use a stand-alone procedure to open the cursor variable. Simply define the REF CURSOR type in a separate package, then reference that type in the stand-alone procedure. For instance, if you create the following (bodiless) package, you can create stand-alone procedures that reference the types it defines:

```
CREATE PACKAGE cv_types AS
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;
   TYPE BonusCurTyp IS REF CURSOR RETURN bonus%ROWTYPE;
   ...
END cv_types;
```

Related Topics

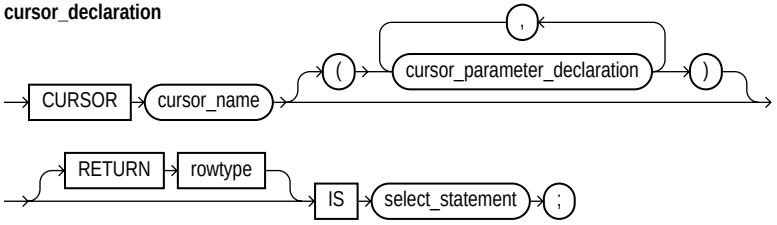
CLOSE Statement, Cursor Attributes, Cursors, FETCH Statement, OPEN-FOR Statement

Cursors

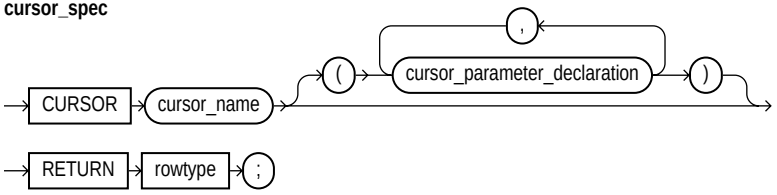
To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. A cursor lets you name the work area, access the information, and process the rows individually. For more information, see "Managing Cursors" on page 5-6.

Syntax

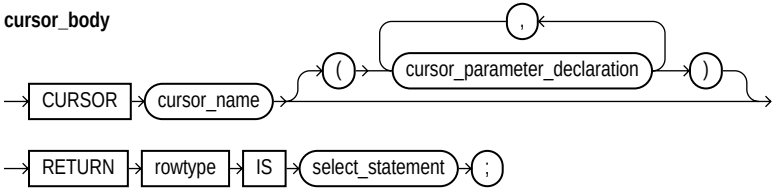
cursor_declaration



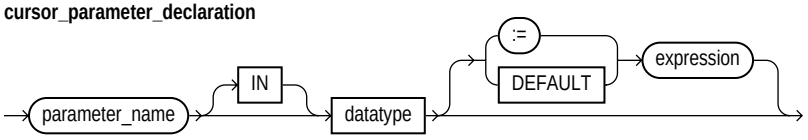
cursor_spec



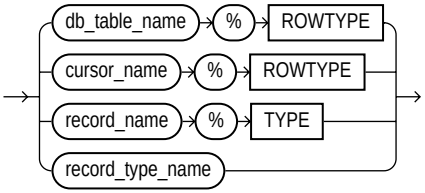
cursor_body



cursor_parameter_declaration



rowtype



Keyword and Parameter Description

select_statement

This is a query that returns a result set of rows. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 11-145. If the cursor declaration declares parameters, each parameter must be used in the query.

RETURN

This keyword introduces the `RETURN` clause, which specifies the datatype of a cursor result 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.

parameter_name

This identifies a cursor parameter; that is, a variable declared as the formal parameter of a cursor. A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be `IN` parameters. The query can also reference other PL/SQL variables within its scope.

db_table_name

This identifies a database table (or view) that must be accessible when the declaration is elaborated.

cursor_name

This identifies an explicit cursor previously declared within the current scope.

record_name

This identifies a user-defined record previously declared within the current scope.

record_type_name

This identifies a `RECORD` type previously defined within the current scope.

%ROWTYPE

This attribute provides a record type that represents a row in a database table or a row fetched from a previously declared cursor. Fields in the record and corresponding columns in the row have the same names and datatypes.

%TYPE

This attribute provides the datatype of a previously declared collection, cursor variable, field, object, record, database column, or variable.

datatype

This is a type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 11-30.

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

Usage Notes

You must declare a cursor before referencing it in an `OPEN`, `FETCH`, or `CLOSE` statement. And, you must declare a variable before referencing it in a cursor declaration. The word SQL is reserved by PL/SQL for use as the default name for implicit cursors and cannot be used in a cursor declaration.

You cannot assign values to a cursor name or use it in an expression. However, cursors and variables follow the same scoping rules. For more information, see "Scope and Visibility" on page 2-37.

You retrieve data from a cursor by opening it, then fetching from it. Because the FETCH statement specifies the target variables, using an INTO clause in the SELECT statement of a cursor_declaration is redundant and invalid.

The scope of cursor parameters is local to the cursor, meaning that they can be referenced only within the query used in the cursor declaration. The values of cursor parameters are used by the associated query when the cursor is opened. The query can also reference other PL/SQL variables within its scope.

The datatype of a cursor parameter must be specified without constraints. For example, the following parameter declarations are illegal:

```
CURSOR c1 (emp_id NUMBER NOT NULL, dept_no NUMBER(2))  -- illegal
```

Examples

Some examples of cursor declarations follow:

```
CURSOR c1 IS SELECT empno, ename, job, sal FROM emp
   WHERE sal > 2000;
CURSOR c2 RETURN dept%ROWTYPE IS
   SELECT * FROM dept WHERE deptno = 10;
CURSOR c3 (start_date DATE) IS
   SELECT empno, sal FROM emp WHERE hiredate > start_date;
```

Related Topics

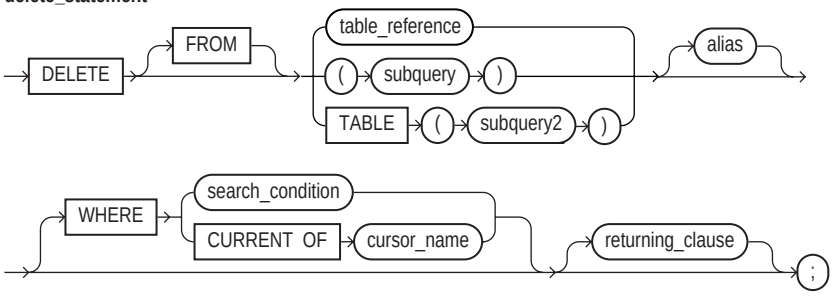
CLOSE Statement, FETCH Statement, OPEN Statement, SELECT INTO Statement

DELETE Statement

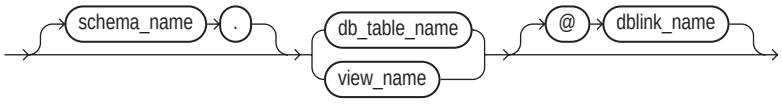
The DELETE statement removes entire rows of data from a specified table or view. For a full description of the DELETE statement, see *Oracle8i SQL Reference*.

Syntax

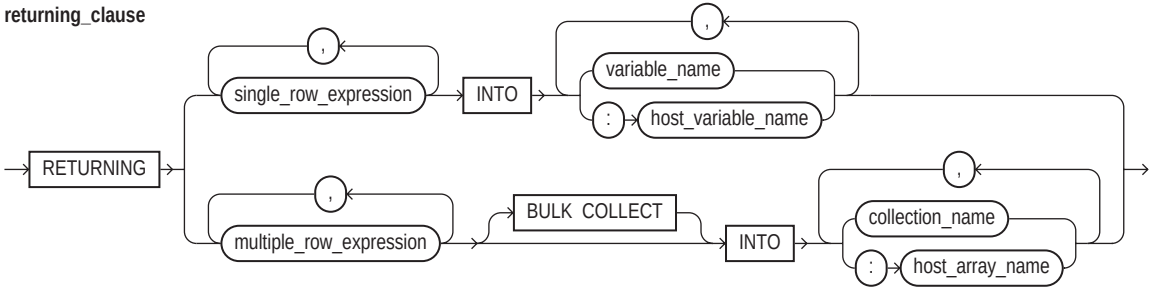
delete_statement



table_reference



returning_clause



Keyword and Parameter Description

table_reference

This specifies a table or view, which must be accessible when you execute the DELETE statement, and for which you must have DELETE privileges.

subquery

This is a SELECT statement that provides a set of rows for processing. Its syntax is like that of select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 11-145.

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 cast as a nested table. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

alias

This is another (usually short) name for the referenced table or view and is typically used in the WHERE clause.

WHERE search_condition

This clause conditionally chooses rows to be deleted from the referenced table or view. Only rows that meet the search condition are deleted. If you omit the WHERE clause, all rows in the table or view are deleted.

WHERE CURRENT OF cursor_name

This clause refers to the latest row processed by the FETCH statement associated with the cursor identified by cursor_name. The cursor must be FOR UPDATE and must be open and positioned on a row. If the cursor is not open, the CURRENT OF clause causes an error.

If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception NO_DATA_FOUND.

returning_clause

This clause lets you return values from the deleted rows, thereby eliminating the need to SELECT the rows beforehand. You can retrieve the column values into variables and/or host variables, or into collections and/or host arrays. However, you cannot use the RETURNING clause for remote or parallel deletes.

BULK COLLECT

This clause instructs the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. The SQL engine bulk-binds all collections referenced in the RETURNING INTO list. The corresponding columns must store scalar (not composite) values. For more information, see "Taking Advantage of Bulk Binds" on page 4-29.

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 SQL cursor and the cursor attributes %NOTFOUND, %FOUND, and %ROWCOUNT let you access useful information about the execution of a DELETE statement.

A DELETE statement might delete one or more rows or no rows. If one or more rows are deleted, you get the following results:

- SQL%NOTFOUND yields FALSE
- SQL%FOUND yields TRUE
- SQL%ROWCOUNT yields the number of rows deleted

If no rows are deleted, you get these results:

- SQL%NOTFOUND yields TRUE
- SQL%FOUND yields FALSE
- SQL%ROWCOUNT yields 0

Examples

The following statement deletes from the bonus table all employees whose sales were below quota:

```
DELETE FROM bonus WHERE sales_amt < quota;
```

The following statement returns column sal from deleted rows and stores the column values in the elements of a host array:

```
DELETE FROM emp WHERE job = 'CLERK' AND sal > 3000
   RETURNING sal INTO :clerk_sals;
```

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 empno = depts(j)
      RETURNING empno BULK COLLECT INTO enums;
```

The column values returned by each execution are added to the values returned previously.

Related Topics

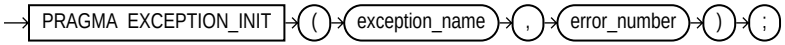
FETCH Statement, SELECT Statement

EXCEPTION_INIT Pragma

The pragma EXCEPTION_INIT associates an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it instead of using the OTHERS handler. For more information, see "Using EXCEPTION_INIT" on page 6-8.

Syntax

exception_init_pragma



Keyword and Parameter Description

PRAGMA

This keyword signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler.

exception_name

This identifies a user-defined exception previously declared within the current scope.

error_number

This is any valid Oracle error number. These are the same error numbers returned by the function SQLCODE.

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

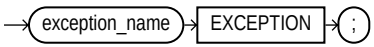
Exceptions, SQLCODE Function

Exceptions

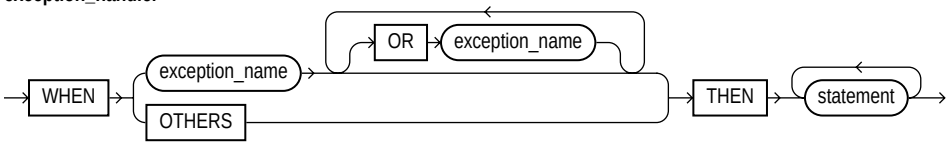
An exception is a runtime error or warning condition, which can be predefined or user-defined. Predefined exceptions are raised implicitly (automatically) by the runtime system. User-defined exceptions must be raised explicitly by `RAISE` statements. To handle raised exceptions, you write separate routines called exception handlers. For more information, see Chapter 6, "Error Handling".

Syntax

exception_declaration



exception_handler



Keyword and Parameter Description

WHEN

This keyword introduces an exception handler. You can have multiple exceptions execute the same sequence of statements by following the keyword `WHEN` with a list of the exceptions, separating them by the keyword `OR`. If any exception in the list is raised, the associated statements are executed.

exception_name

This identifies a predefined exception such as `ZERO_DIVIDE`, or a user-defined exception previously declared within the current scope.

OTHERS

This keyword stands for all the exceptions not explicitly named in the exception-handling part of the block. The use of `OTHERS` is optional and is allowed only as the last exception handler. You cannot include `OTHERS` in a list of exceptions following the keyword `WHEN`.

statement

This is an executable statement. For the syntax of `statement`, see "Blocks" on page 11-7.

Usage Notes

An exception declaration can appear only in the declarative part of a block, subprogram, or package. The scope rules for exceptions and variables are the same. But, unlike variables, exceptions cannot be passed as parameters to subprograms.

Some exceptions are predefined by PL/SQL. For a list of these exceptions, see "Predefined Exceptions" on page 6-4. 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 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.

However, exceptions cannot propagate across remote procedure calls (RPCs). Therefore, a PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see "Using raise_application_error" on page 6-9.

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.

An exception handler can reference only those variables that the current block can reference.

Example

The following PL/SQL block has two exception handlers:

```
DELARE
   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;
```

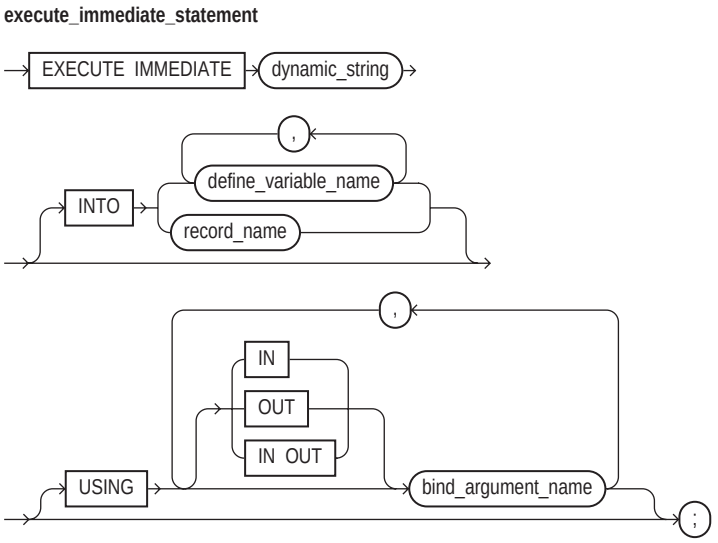
Related Topics

Blocks, EXCEPTION_INIT Pragma, RAISE Statement

EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes a dynamic SQL statement or an anonymous PL/SQL block. For more information, see Chapter 10, "Native Dynamic SQL".

Syntax



Keyword and Parameter Description

dynamic_string

This is a string literal, variable, or expression that represents a SQL statement or PL/SQL block.

define_variable_name

This identifies a variable that stores a SELECTed column value.

record_name

This identifies a user-defined or %ROWTYPE record that stores a SELECTed row.

bind_argument_name

This identifies an expression whose value is passed to the dynamic SQL statement or PL/SQL block.

INTO ...

This clause, useful only for single-row queries, specifies the variables or record into which column values are fetched.

USING ...

This clause specifies a list of bind arguments. If you do not specify a parameter mode, it defaults to `IN`. At run time, any bind arguments in the `USING` clause replace corresponding placeholders in the SQL statement or PL/SQL block.

Usage Notes

Except for multi-row queries, the dynamic string can contain any SQL statement (*without* the terminator) or any PL/SQL block (with the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement. For the right way, see "Passing the Names of Schema Objects" on page 10-10.

For each column value returned by the query, there must be a corresponding, type-compatible variable or field in the `INTO` clause Also, every placeholder in the dynamic string must be associated with a bind argument in the `USING` clause.

Numeric, character, and string literals are allowed in the `USING` clause, but Boolean literals (`TRUE`, `FALSE`, `NULL`) are not. To pass nulls to the dynamic string, you must use a workaround (see "Passing Nulls" on page 10-12).

Dynamic SQL supports all the SQL datatypes. So, for example, define variables and bind arguments can be collections, `LOB`s, instances of an object type, and refs. As a rule, dynamic SQL does not support PL/SQL-specific types. So, for example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the `INTO` clause.

You can execute a dynamic SQL statement repeatedly using new values for the bind arguments. However, you incur some overhead because `EXECUTE IMMEDIATE` re-prepares the dynamic string before every execution.

Examples

The following PL/SQL block contains several examples of dynamic SQL:

```
DECLARE
   sql_stmt    VARCHAR2(100);
   plsql_block VARCHAR2(200);
   my_deptno   NUMBER(2) := 50;
   my_dname    VARCHAR2(15) := 'PERSONNEL';
   my_loc      VARCHAR2(15) := 'DALLAS';
   emp_rec     emp%ROWTYPE;
BEGIN
   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING my_deptno, my_dname, my_loc;

   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING 7788;

   EXECUTE IMMEDIATE 'DELETE FROM dept
      WHERE deptno = :n' USING my_deptno;

   plsql_block := 'BEGIN emp_stuff.raise_salary(:id, :amt); END;';
   EXECUTE IMMEDIATE plsql_block USING 7788, 500;

   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';

   sql_stmt := 'ALTER SESSION SET SQL_TRACE TRUE';
   EXECUTE IMMEDIATE sql_stmt;
END;
```

Related Topics

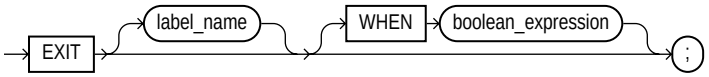
OPEN-FOR-USING Statement

EXIT Statement

You use the EXIT statement to exit a loop. The EXIT statement has two forms: the unconditional EXIT and the conditional EXIT WHEN. With either form, you can name the loop to be exited. For more information, see "Iterative Control: LOOP and EXIT Statements" on page 3-6.

Syntax

exit_statement



Keyword and Parameter Description

EXIT

An unconditional EXIT statement (that is, one without a WHEN clause) exits the current loop immediately. Execution resumes with the statement following the loop.

label_name

This identifies the loop to be exited. You can exit not only the current loop but any enclosing labeled loop.

boolean_expression

This is an expression that yields the Boolean value TRUE, FALSE, or NULL. It is evaluated with each iteration of the loop in which the EXIT WHEN statement appears. If the expression yields TRUE, the current loop (or the loop labeled by label_name) is exited immediately. For the syntax of boolean_expression, see "Expressions" on page 11-63.

Usage Notes

The EXIT statement can be used only inside a loop. PL/SQL allows you to 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 illegal 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;  -- illegal
      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
   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

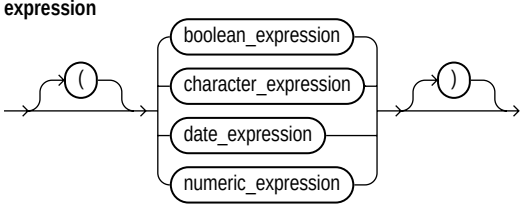
Expressions

An expression is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression is a single variable.

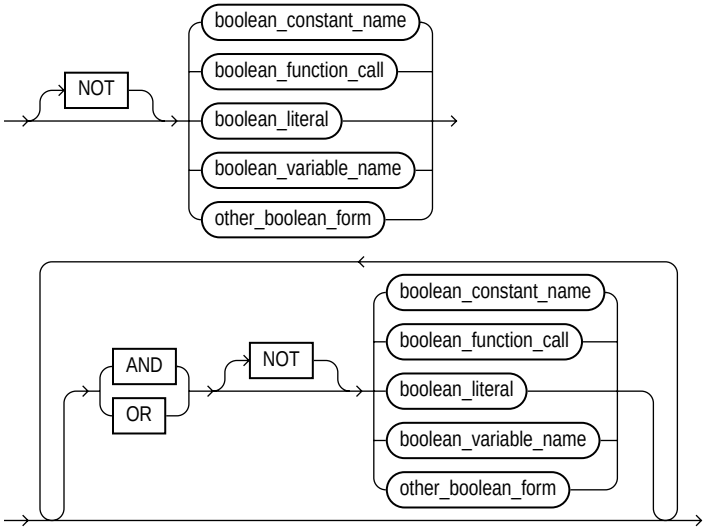
The PL/SQL compiler determines the datatype of an expression from the types of the variables, constants, literals, and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results. For more information, see "Expressions and Comparisons" on page 2-41.

Syntax

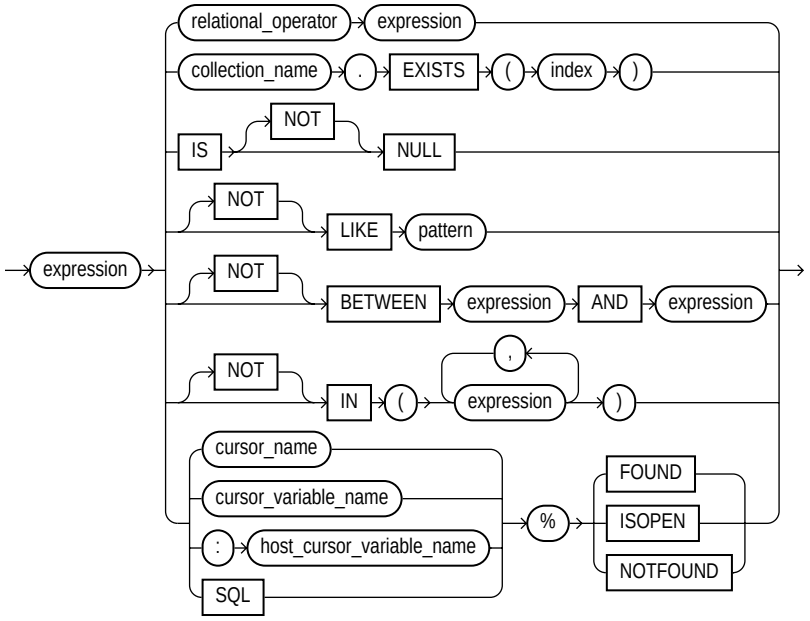
expression



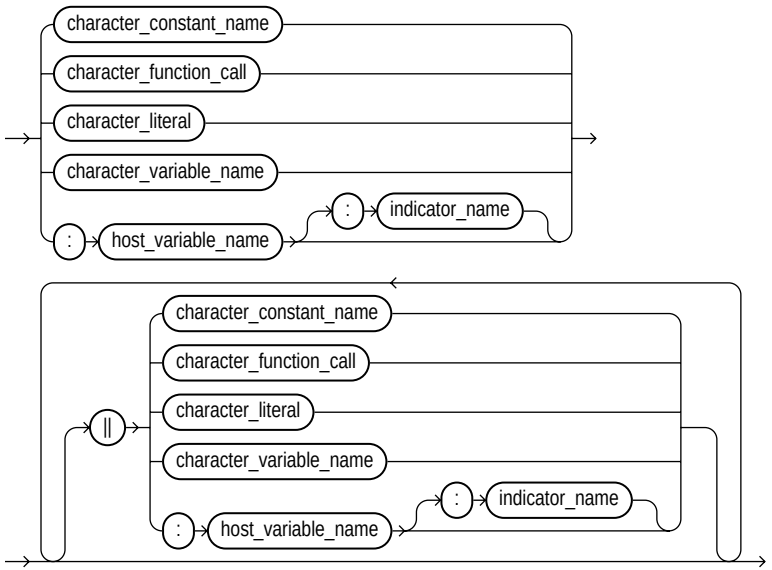
boolean_expression



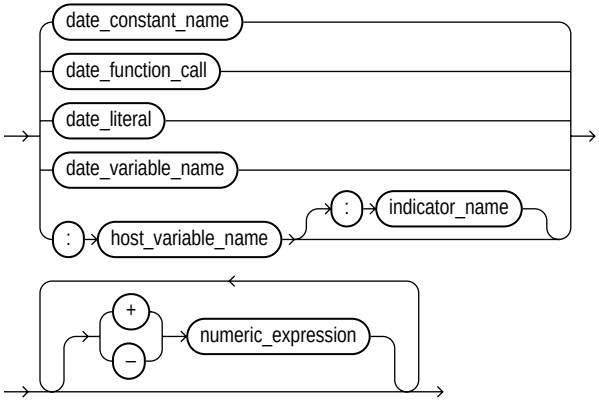
other_boolean_form



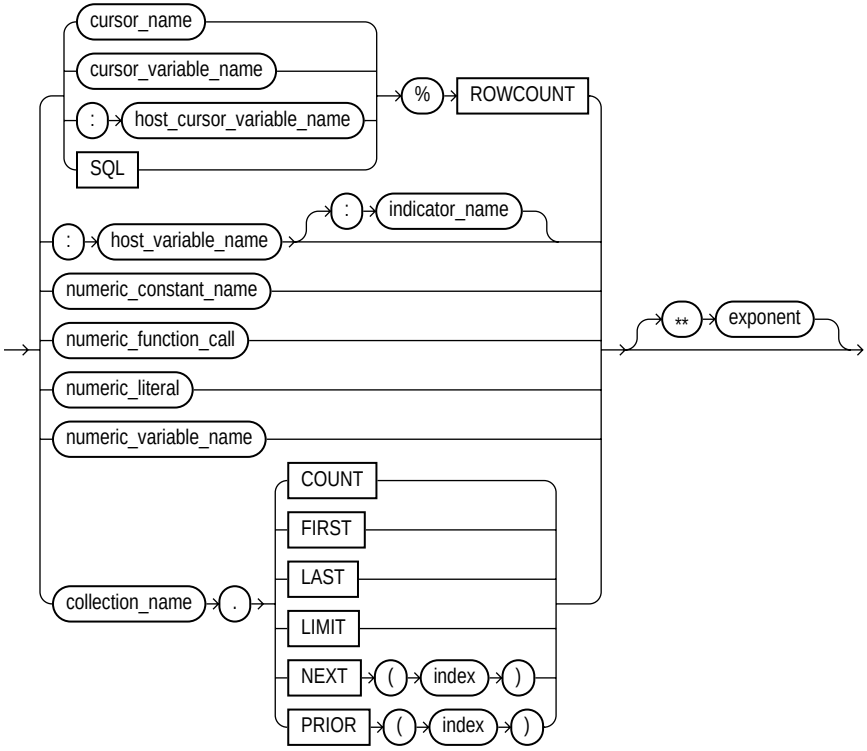
character_expression

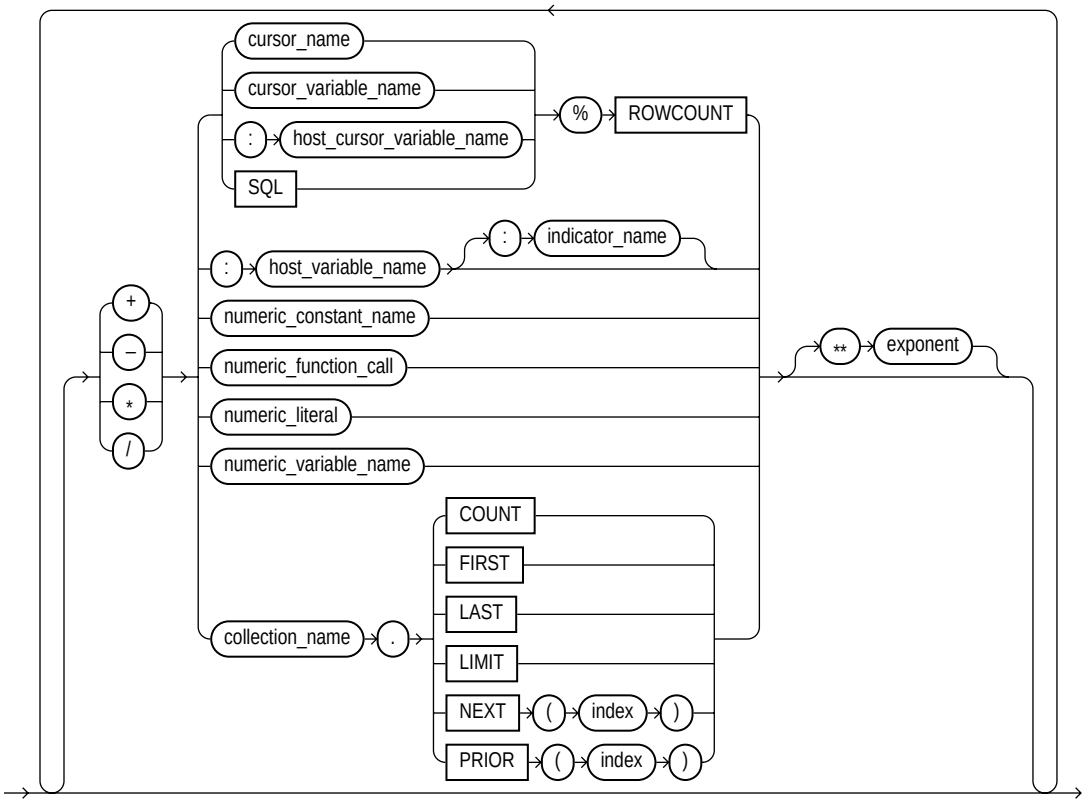


date_expression



numeric_expression





Keyword and Parameter Description

boolean_expression

This is an expression that yields the Boolean value TRUE, FALSE, or NULL.

character_expression

This is an expression that yields a character or character string.

date_expression

This is an expression that yields a date/time value.

numeric_expression

This is an expression that yields an integer or real value.

NOT, AND, OR

These are logical operators, which follow the tri-state logic of Table 2–3 on page 2-43. 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. NOT NULL returns NULL because nulls are indeterminate. For more information, see "Logical Operators" on page 2-43.

boolean_constant_name

This identifies a constant of type BOOLEAN, which must be initialized to the value TRUE, FALSE, or NULL. Arithmetic operations on Boolean constants are illegal.

boolean_function_call

This is any function call that returns a Boolean value.

boolean_literal

This is the predefined value TRUE, FALSE, or NULL (which stands for a missing, unknown, or inapplicable value). You cannot insert the value TRUE or FALSE into a database column.

boolean_variable_name

This identifies a variable of type BOOLEAN. Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable. You cannot select or fetch column values into a BOOLEAN variable. Also, arithmetic operations on BOOLEAN variables are illegal.

relational_operator

This operator allows you to compare expressions. For the meaning of each operator, see "Comparison Operators" on page 2-44.

IS [NOT] NULL

This comparison operator returns the Boolean value TRUE if its operand is null, or FALSE if its operand is not null.

[NOT] LIKE

This comparison operator compares a character value to a pattern. Case is significant. LIKE returns the Boolean value TRUE if the character patterns match, or FALSE if they do not match.

pattern

This is a character string compared by the LIKE operator to a specified string value. It can include two special-purpose characters called wildcards. An underscore (_) matches exactly one character; a percent sign (%) matches zero or more characters.

[NOT] 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*."

[NOT] IN

This comparison operator tests set membership. It means "equal to any member of." The set can contain nulls, but they are ignored. Also, expressions of the form

```
value NOT IN set
```

yield FALSE if the set contains a null.

cursor_name

This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable previously declared within the current scope.

host_cursor_variable_name

This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Host cursor variables must be prefixed with a colon.

SQL

This identifies a cursor opened implicitly by Oracle to process a SQL data manipulation statement. The implicit SQL cursor always refers to the most recently executed SQL statement.

%FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT

These are cursor attributes. When appended to the name of a cursor or cursor variable, these attributes return useful information about the execution of a multi-row query. You can also append them to the implicit SQL cursor. For more information, see "Using Cursor Attributes" on page 5-34.

EXISTS, COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR

These are collection methods. When appended to the name of a collection, these methods return useful information. For example, `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 11-16.

index

This is a numeric expression that must yield a value of type `BINARY_INTEGER` or a value implicitly convertible to that datatype.

host_variable_name

This identifies a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host variable must be implicitly convertible to the appropriate PL/SQL datatype. Also, host variables must be prefixed with a colon.

indicator_name

This identifies an indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable "indicates" the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables can detect nulls or truncated values in output host variables.

numeric_constant_name

This identifies a previously declared constant that stores a numeric value. It must be initialized to a numeric value or a value implicitly convertible to a numeric value.

numeric_function_call

This is a function call that returns a numeric value or a value implicitly convertible to a numeric value.

numeric_literal

This is a literal that represents a numeric value or a value implicitly convertible to a numeric value.

collection_name

This identifies a nested table, index-by table, or varray previously declared within the current scope.

numeric_variable_name

This identifies a previously declared variable that stores a numeric value.

NULL

This keyword represents a null; it stands for a missing, unknown, or inapplicable value. When NULL is used in a numeric or date expression, the result is a null.

exponent

This is an expression that must yield a numeric value.

+, -, /, *, **

These symbols are the addition, subtraction, division, multiplication, and exponentiation operators, respectively.

character_constant_name

This identifies a previously declared constant that stores a character value. It must be initialized to a character value or a value implicitly convertible to a character value.

character_function_call

This is a function call that returns a character value or a value implicitly convertible to a character value.

character_literal

This is a literal that represents a character value or a value implicitly convertible to a character value.

character_variable_name

This identifies a previously declared variable that stores a character value.

||

This is the concatenation operator. As the following example shows, the result of concatenating *string1* with *string2* is a character string that contains *string1* followed by *string2*:

```
'Good' || ' morning!' = 'Good morning!'
```

The next example shows that nulls have no effect on the result of a concatenation:

```
'suit' || NULL || 'case' = 'suitcase'
```

A null string (`''`), which is zero characters in length, is treated like a null.

date_constant_name

This identifies a previously declared constant that stores a date value. It must be initialized to a date value or a value implicitly convertible to a date value.

date_function_call

This is a function call that returns a date value or a value implicitly convertible to a date value.

date_literal

This is a literal that represents a date value or a value implicitly convertible to a date value.

date_variable_name

This identifies a previously declared variable that stores a date value.

Usage Notes

In a Boolean expression, you can only compare values that have compatible datatypes. For more information, see "Datatype Conversion" on page 2-27.

In conditional control statements, if a Boolean expression yields TRUE, its associated sequence of statements is executed. But, if the expression yields FALSE or NULL, its associated sequence of statements is *not* executed.

The relational operators can be applied to operands of type BOOLEAN. By definition, TRUE is greater than FALSE. Comparisons involving nulls always yield a null. The value of a Boolean expression can be assigned only to Boolean variables, not to host variables or database columns. Also, datatype conversion to or from type BOOLEAN is not supported.

You can use the addition and subtraction operators to increment or decrement a date value, as the following examples show:

```
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 value 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

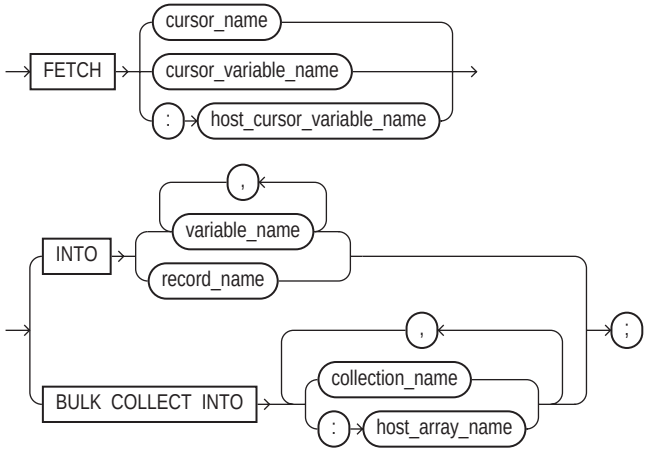
Assignment Statement, Constants and Variables, EXIT Statement, IF Statement, LOOP Statements

FETCH Statement

The FETCH statement retrieves rows of data one at a time from the result set of a multi-row query. The data is stored in variables or fields that correspond to the columns selected by the query. For more information, see "Managing Cursors" on page 5-6.

Syntax

fetch_statement



Keyword and Parameter Description

cursor_name

This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope.

host_cursor_variable_name

This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

BULK COLLECT

This clause instructs the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. The SQL engine bulk-binds all collections referenced in the `INTO` list. The corresponding columns must store scalar (not composite) values. For more information, see "Taking Advantage of Bulk Binds" on page 4-29.

variable_name

This identifies a previously declared scalar 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.

record_name

This identifies a user-defined or `%ROWTYPE` record into which rows of values are fetched. For each column value returned by the query associated with the cursor or cursor variable, there must be a corresponding, type-compatible field in the record.

collection_name

This identifies a declared collection into which column values are bulk fetched. For each query `select_item`, there must be a corresponding, type-compatible collection in the list.

host_array_name

This identifies an array (declared in a PL/SQL host environment and passed to PL/SQL as a bind variable) into which column values are bulk fetched. For each query `select_item`, there must be a corresponding, type-compatible array in the list. Host arrays must be prefixed with a colon.

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 yields 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.

Eventually, the FETCH statement must fail to return a row, so when that happens, no exception is raised. To detect the failure, you must use the cursor attribute %FOUND or %NOTFOUND. For more information, see "Using Cursor Attributes" on page 5-34.

PL/SQL raises the predefined exception INVALID_CURSOR if you try to fetch from a closed or never-opened cursor or cursor variable.

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;
```

In the following Pro*C example, you fetch rows from a host cursor variable into a host record (struct) named `emp_rec`:

```
/* Exit loop when done fetching. */
EXEC SQL WHENEVER NOTFOUND DO break;
for (;;)
{
   /* Fetch row into record. */
   EXEC SQL FETCH :emp_cur INTO :emp_rec;
}
```

The next example shows that 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.

```
for (;;)
{
   /* Fetch row from result set. */
   EXEC SQL FETCH :emp_cur INTO :emp_rec1;
   /* Fetch next row from same result set. */
   EXEC SQL FETCH :emp_cur INTO :emp_rec2;
}
```

Related Topics

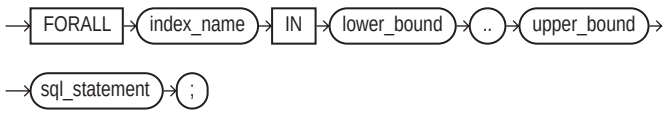
CLOSE Statement, Cursors, Cursor Variables, LOOP Statements, OPEN Statement, OPEN-FOR Statement

FORALL Statement

The FORALL statement instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is *not* a FOR loop. For more information, see "Taking Advantage of Bulk Binds" on page 4-29.

Syntax

forall_statement



Keyword and Parameter Description

index_name

This is an undeclared identifier that can be referenced only within the FORALL statement and only as a collection subscript.

The implicit declaration of index_name overrides any other declaration outside the loop. So, another variable with the same name cannot be referenced inside the statement. Inside a FORALL statement, index_name cannot appear in expressions and cannot be assigned a value.

lower_bound .. upper_bound

These are expressions that must yield integer values and must specify a valid range of consecutive index numbers. The expressions are evaluated only when the FORALL statement is first entered.

The index is assigned the value of lower_bound. If that value is not greater than the value of upper_bound, the SQL statement is executed, then the index is incremented. If the value of the index is still not greater than the value of upper_bound, the SQL statement is executed again. This process repeats until the value of the index is greater than the value of upper_bound. At that point, the FORALL statement completes.

sql_statement

This must be an INSERT, UPDATE, or DELETE statement that references collection elements.

Usage Notes

The SQL statement can reference more than one collection. However, the PL/SQL engine bulk-binds only subscripted collections.

All collection elements in the specified range must exist. If an element is missing or was deleted, you get an error.

If a FORALL statement fails, database changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous executions are *not* rolled back.

Example

The following example shows that you can use the lower and upper bounds to bulk-bind arbitrary slices of a collection:

```
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;
```

Remember, the PL/SQL engine bulk-binds only subscripted collections. So, in the following example, it does not bulk-bind the collection sals, which is passed to the function median:

```
FORALL i IN 1..20
   INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);
```

Related Topics

BULK COLLECT Clause

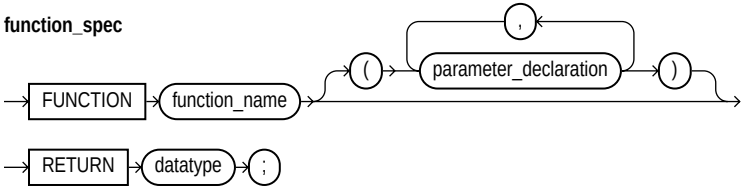
Functions

A function is a subprogram that can take parameters and be invoked. Generally, you use a function to compute a value. A function has two parts: the specification and the body. The specification (spec for short) begins with the keyword `FUNCTION` and ends with the `RETURN` clause, which specifies the datatype of the result value. Parameter declarations are optional. Functions that take no parameters are written without parentheses. The function body begins with the keyword `IS` and ends with the keyword `END` followed by an optional function name.

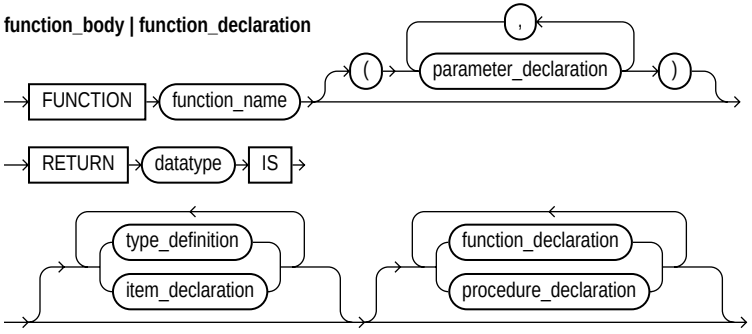
The function body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. These items are local and cease to exist when you exit the function. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution. For more information, see "Functions" on page 7-5.

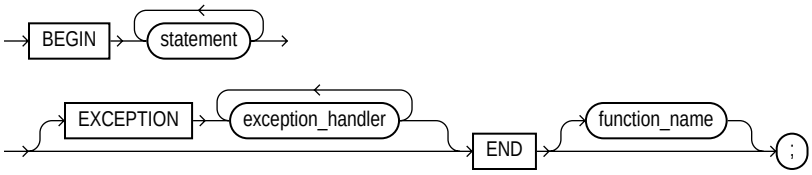
Syntax

function_spec

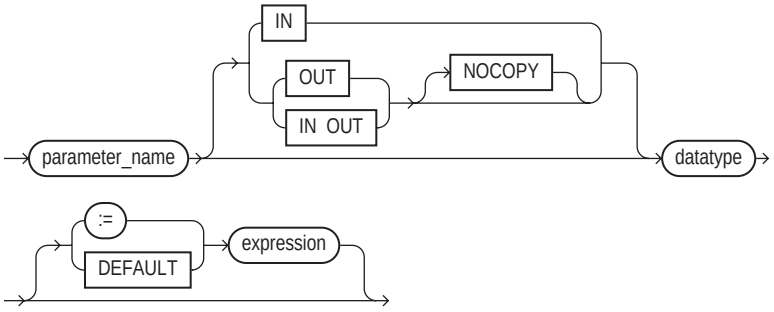


function_body | function_declaration





parameter_declaration



Keyword and Parameter Description

function_name

This identifies a user-defined function.

parameter_name

This identifies a formal parameter, which is a variable declared in a function spec and referenced in the function body.

IN, OUT, IN OUT

These parameter modes define the behavior of formal parameters. An `IN` parameter lets you pass values to the subprogram being called. An `OUT` parameter lets you return values to the caller of the subprogram. An `IN OUT` parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

NOCOPY

This is a compiler hint (not directive), which allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference instead of by value (the default). For more information, see "NOCOPY Compiler Hint" on page 7-17.

datatype

This is a type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 11-30.

:= | DEFAULT

This operator or keyword allows you to initialize `IN` parameters to default values.

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

RETURN

This keyword introduces the `RETURN` clause, which specifies the datatype of the result value.

type_definition

This specifies a user-defined datatype. For the syntax of `type_definition`, see "Blocks" on page 11-7.

item_declaration

This declares a program object. For the syntax of `item_declaration`, see "Blocks" on page 11-7.

function_declaration

This construct declares a function. For the syntax of `function_declaration`, see "Functions" on page 11-79.

procedure_declaration

This construct declares a procedure. For the syntax of `procedure_declaration`, see "Procedures" on page 11-127.

exception_handler

This construct 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 11-55.

Usage Notes

A function is called as part of an expression. For example, the function `sal_ok` might be called as follows:

```
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

Every function must contain at least one `RETURN` statement. Otherwise, PL/SQL raises the predefined exception `PROGRAM_ERROR` at run time.

To be callable from SQL statements, a stored function must obey certain rules meant to control side effects. See "Controlling Sides Effects" on page 7-7.

You can write the function spec and body as a unit. Or, you can separate the function spec from its body. That way, you can hide implementation details by placing the function in a package. You can define functions in a package body without declaring their specs in the package spec. However, such functions can be called only from inside the package.

Inside a function, an `IN` parameter acts like a constant. So, you cannot assign it a value. An `OUT` parameter acts like a local variable. So, you can change its value and reference the value in any way. An `IN OUT` parameter acts like an initialized variable. So, you can assign it a value, which can be assigned to another variable. For summary information about the parameter modes, see Table 7–1 on page 7-16.

Avoid using the `OUT` and `IN OUT` modes with functions. The purpose of a function is to take zero or more parameters and return a single value. Also, functions should be free from side effects, which change the values of variables not local to the subprogram.

Functions can be defined using any Oracle tool that supports PL/SQL. However, to become available for general use, functions must be `CREATE`d and stored in an Oracle database. You can issue the `CREATE FUNCTION` statement interactively from SQL*Plus.

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

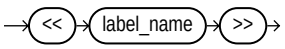
Collections, Packages, Procedures, Records

GOTO Statement

The GOTO statement branches unconditionally to a statement label or block label. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. The GOTO statement transfers control to the labelled statement or block. For more information, see "GOTO Statement" on page 3-15.

Syntax

label_declaration



goto_statement

Keyword and Parameter Description

label_name

This is an undeclared identifier that labels an executable statement or a PL/SQL block. You use a GOTO statement to transfer control to the statement or block following <<label_name>>.

Usage Notes

Some possible destinations of a GOTO statement are illegal. In particular, a GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block. For example, the following GOTO statement is illegal:

```
BEGIN
   ...
   GOTO update_row;  -- illegal branch into IF statement
   ...
   IF valid THEN
      ...
      <<update_row>>
      UPDATE emp SET ...
   END IF;
```

From the current block, a GOTO statement can branch to another place in the block or into an enclosing block, but not into an exception handler. From an exception handler, a GOTO statement can branch into an enclosing block, but not into the current block.

If you use the GOTO statement to exit a cursor FOR loop prematurely, the cursor is closed automatically. The cursor is also closed automatically if an exception is raised inside the loop.

A given label can appear only once in a block. However, the label can appear in other blocks including enclosing blocks and sub-blocks. If a GOTO statement cannot find its target label in the current block, it branches to the first enclosing block in which the label appears.

Examples

A GOTO label cannot precede just any keyword. It must precede an executable statement or a PL/SQL block. For example, the following GOTO statement is illegal:

```
FOR ctr IN 1..50 LOOP
   DELETE FROM emp WHERE ...
   IF SQL%FOUND THEN
      GOTO end_loop;  -- illegal
   END IF;
   ...
<<end_loop>>
END LOOP;   -- not an executable statement
```

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

```
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;
```

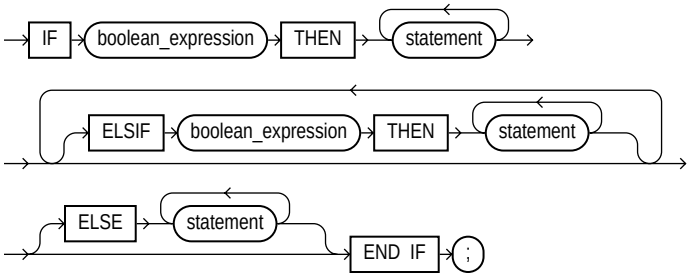
For more examples of legal and illegal GOTO statements, see "GOTO Statement" on page 3-15.

IF Statement

The IF statement lets you execute a sequence of statements conditionally. Whether the sequence is executed or not depends on the value of a Boolean expression. For more information, see "Conditional Control: IF Statements" on page 3-2.

Syntax

if_statement



Keyword and Parameter Description

boolean_expression

This is an expression that yields the Boolean value TRUE, FALSE, or NULL. It is associated with a sequence of statements, which is executed only if the expression yields TRUE.

THEN

This keyword associates the Boolean expression that precedes it with the sequence of statements that follows it. If the expression yields TRUE, the associated sequence of statements is executed.

ELSIF

This keyword introduces a Boolean expression to be evaluated if the expression following IF and all the expressions following any preceding ELSIFs yield FALSE or NULL.

ELSE

If control reaches this keyword, the sequence of statements that follows it is executed.

Usage Notes

There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The simplest form of IF statement associates a Boolean expression with a sequence of statements enclosed by the keywords THEN and END IF. The sequence of statements is executed only if the expression yields TRUE. If the expression yields FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements. The sequence of statements in the ELSE clause is executed only if the Boolean expression yields FALSE or NULL. Thus, the ELSE clause ensures that a sequence of statements is executed.

The third form of IF statement uses the keyword ELSIF to introduce additional Boolean expressions. If the first expression yields FALSE or NULL, the ELSIF clause evaluates another expression. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Boolean expressions are evaluated one by one from top to bottom. If any expression yields TRUE, its associated sequence of statements is executed and control passes to the next statement. If all expressions yield FALSE or NULL, the sequence in the ELSE clause is executed.

An IF statement never executes more than one sequence of statements because processing is complete after any sequence of statements is executed. However, the THEN and ELSE clauses can include more IF statements. That is, IF statements can be nested.

Examples

In the example below, if shoe_count has a value of 10, both the first and second Boolean expressions yield TRUE. Nevertheless, order_quantity is assigned the proper value of 50 because processing of an IF statement stops after an expression yields TRUE and its associated sequence of statements is executed. The expression associated with ELSIF is never evaluated and control passes to the INSERT statement.

```
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;
```

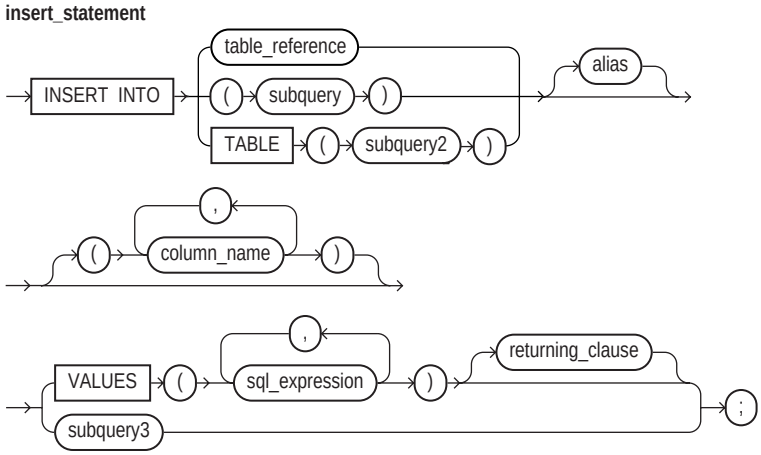
Related Topics

Expressions

INSERT Statement

The INSERT statement adds new rows of data to a specified database table or view. For a full description of the INSERT statement, see *Oracle8i SQL Reference*.

Syntax



Keyword and Parameter Description

table_reference

This identifies a table or view that must be accessible when you execute the INSERT statement, and for which you must have INSERT privileges. For the syntax of table_reference, see "DELETE Statement" on page 11-49.

subquery

This is a SELECT statement that provides a set of rows for processing. Its syntax is like that of select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 11-145.

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 cast as a nested table. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

alias

This is another (usually short) name for the referenced table or view.

column_name[, column_name]...

This identifies a list of columns in a database table or view. Column names need not appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement. However, no column name can appear more than once in the list. If the list does not include all the columns in a table, the missing columns are set to NULL or to a default value specified in the CREATE TABLE statement.

sql_expression

This is any expression valid in SQL. For more information, see *Oracle8i SQL Reference*.

VALUES (...)

This clause assigns the values of expressions to corresponding columns in the column list. If there is no column list, the first value is inserted into the first column defined by the CREATE TABLE statement, the second value is inserted into the second column, and so on.

There must be only one value for each column in the column list. The first value is associated with the first column, the second value is associated with the second column, and so on. If there is no column list, you must supply a value for each column in the table.

The datatypes of the values being inserted must be compatible with the datatypes of corresponding columns in the column list.

As many rows are added to the table as are returned by the subquery in the VALUES clause. 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.

subquery3

This is a SELECT statement that provides a value or set of values to the VALUES clause. The subquery must return only one row containing a value for every column in the column list or for every column in the table if there is no column list.

returning_clause

This clause lets you return values from inserted rows, thereby eliminating the need to SELECT the rows afterward. You can retrieve the column values into variables and/or host variables, or into collections and/or host arrays. However, you cannot use the RETURNING clause for remote or parallel inserts. For the syntax of `returning_clause`, see "DELETE Statement" on page 11-49.

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 SQL cursor and cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of an INSERT statement.

An INSERT statement might insert one or more rows or no rows. If one or more rows are inserted, you get the following results:

- SQL%NOTFOUND yields FALSE
- SQL%FOUND yields TRUE
- SQL%ROWCOUNT yields the number of rows inserted

If no rows are inserted, you get these results:

- SQL%NOTFOUND yields TRUE
- SQL%FOUND yields FALSE
- SQL%ROWCOUNT yields 0

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

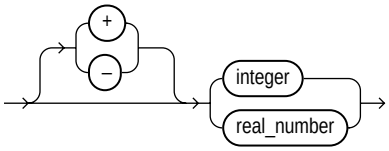
SELECT Statement

Literals

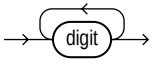
A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. The numeric literal 135 and the string literal `'hello world'` are examples. For more information, see "Literals" on page 2-7.

Syntax

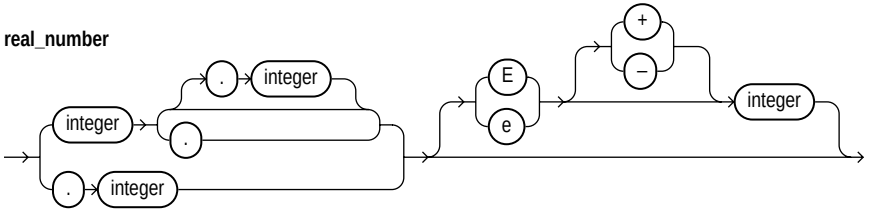
numeric_literal



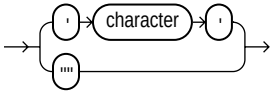
integer



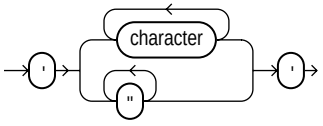
real_number



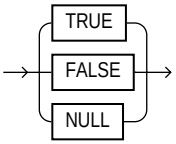
character_literal



string_literal



boolean_literal



Keyword and Parameter Description

integer

This is an optionally signed whole number without a decimal point.

real_number

This is an optionally signed whole or fractional number with a decimal point.

digit

This is one of the numerals 0 .. 9.

char

This is a member of the PL/SQL character set. For more information, see "Character Set" on page 2-2.

TRUE, FALSE, NULL

This is a predefined Boolean value.

Usage Notes

Two kinds of numeric literals can be used in arithmetic expressions: integers and reals. Numeric literals must be separated by punctuation. Spaces can be used in addition to the punctuation.

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

PL/SQL is case sensitive within character literals. So, for example, PL/SQL considers the literals `'Q'` and `'q'` to be different.

A string literal is a sequence of zero or more characters enclosed by single quotes. The null string (`''`) contains zero characters. To represent an apostrophe within a string, write two single quotes. PL/SQL is case sensitive within string literals. So, for example, PL/SQL considers the literals `'white'` and `'White'` to be different.

Also, trailing blanks are significant within string literals, so `'abc'` and `'abc '` are different. Trailing blanks in a literal are never trimmed.

The Boolean values TRUE and FALSE cannot be inserted into a database column.

Examples

Several examples of numeric literals follow:

```
25   6.34   7E2   25e-03   .1   1.   +17  -4.4
```

Several examples of character literals follow:

```
'H'   '&'   ' '   '9'   ']'   'g'
```

A few examples of string literals follow:

```
'$5,000'
'02-AUG-87'
'Don''t leave without saving your work.'
```

Related Topics

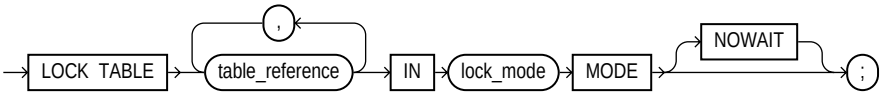
Constants and Variables, Expressions

LOCK TABLE Statement

The LOCK TABLE statement lets you lock entire database tables in a specified lock mode. That lets you can share or deny access to the tables while maintaining their integrity. For more information, see "Using LOCK TABLE" on page 5-48.

Syntax

lock_table_statement



Keyword and Parameter Description

table_reference

This identifies a table or view that must be accessible when you execute the LOCK TABLE statement. For the syntax of `table_reference`, see "DELETE Statement" on page 11-49.

lock_mode

This parameter specifies the lock mode. It must be one of the following: ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE, SHARE, SHARE ROW EXCLUSIVE, or EXCLUSIVE.

NOWAIT

This optional keyword tells Oracle not to wait if the table has been locked by another user. Control is immediately returned to your program, so it can do other work before trying again to acquire the lock.

Usage Notes

If you omit the keyword NOWAIT, Oracle waits until the table is available; the wait has no set limit. Table locks are released when your transaction issues a commit or rollback.

A table lock never keeps other users from querying a table, and a query never acquires a table lock.

If your program includes SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as SELECT, INSERT, UPDATE, or DELETE.

Example

The following statement locks the accts table in shared mode:

```
LOCK TABLE accts IN SHARE MODE;
```

Related Topics

COMMIT Statement, ROLLBACK Statement, UPDATE Statement

LOOP Statements

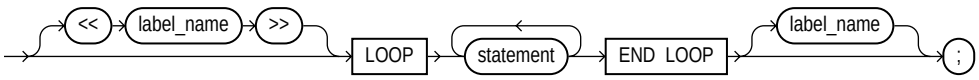
LOOP statements execute a sequence of statements multiple times. The loop encloses the sequence of statements that is to be repeated. PL/SQL provides the following types of loop statements:

- basic loop
- WHILE loop
- FOR loop
- cursor FOR loop

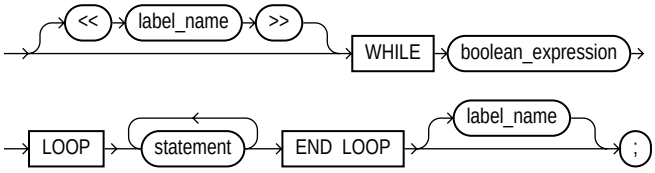
For more information, see "Iterative Control: LOOP and EXIT Statements" on page 3-6.

Syntax

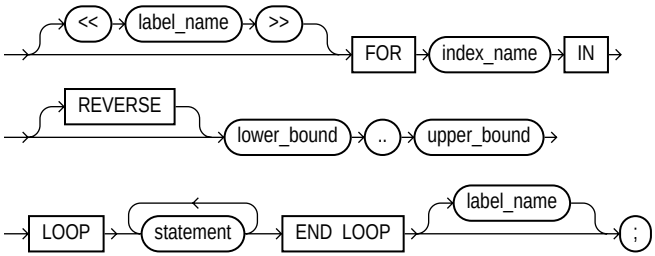
basic_loop_statement



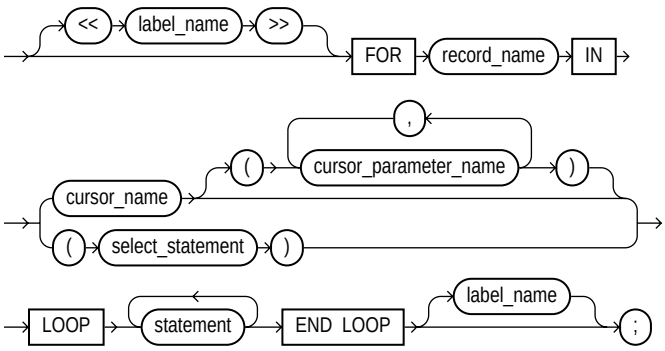
while_loop_statement



for_loop_statement



cursor_for_loop_statement



Keyword and Parameter Description

label_name

This is an undeclared identifier that optionally labels a loop. If used, `label_name` must be enclosed by double angle brackets and must appear at the beginning of the loop. Optionally, `label_name` (*not* enclosed in angle brackets) can also appear at the end of the loop.

You can use `label_name` in an `EXIT` statement to exit the loop labelled by `label_name`. You can exit not only the current loop, but any enclosing loop.

You cannot reference the index of a `FOR` loop from a nested `FOR` loop if both indexes have the same name unless the outer loop is labeled by `label_name` and you use dot notation, as follows:

```
label_name.index_name
```

In the following example, you compare two loop indexes that have the same name, one used by an enclosing loop, the other by a nested loop:

```
<<outer>>
FOR ctr IN 1..20 LOOP
   ...
   <<inner>>
   FOR ctr IN 1..10 LOOP
      IF outer.ctr > ctr THEN ...
   END LOOP inner;
END LOOP outer;
```

basic_loop_statement

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP. With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use the EXIT, GOTO, or RAISE statement to complete the loop. A raised exception will also complete the loop.

while_loop_statement

The WHILE-LOOP statement associates a Boolean expression with a sequence of statements enclosed by the keywords LOOP and END LOOP. Before each iteration of the loop, the expression is evaluated. If the expression yields TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the expression yields FALSE or NULL, the loop is bypassed and control passes to the next statement.

boolean_expression

This is an expression that yields the Boolean value TRUE, FALSE, or NULL. It is associated with a sequence of statements, which is executed only if the expression yields TRUE. For the syntax of boolean_expression, see "Expressions" on page 11-63.

for_loop_statement

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. Numeric FOR loops iterate over a specified range of integers. (Cursor FOR loops, which iterate over the result set of a cursor.) The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

The range is evaluated when the FOR loop is first entered and is never re-evaluated. The sequence of statements in the loop is executed once for each integer in the range defined by lower_bound..upper_bound. After each iteration, the loop index is incremented.

index_name

This is an undeclared identifier that names the loop index (sometimes called a loop counter). Its scope is the loop itself. Therefore, you cannot reference the index outside the loop.

The implicit declaration of index_name overrides any other declaration outside the loop. So, another variable with the same name cannot be referenced inside the loop unless a label is used, as follows:

```
<<main>>
DECLARE
   num NUMBER;
BEGIN
   ...
   FOR num IN 1..10 LOOP
      ...
      IF main.num > 5 THEN  -- refers to the variable num,
         ...                -- not to the loop index
      END IF;
   END LOOP;
END main;
```

Inside a loop, its index is treated like a constant. The index can appear in expressions, but cannot be assigned a value.

lower_bound .. upper_bound

These are expressions that must yield integer values. The expressions are evaluated only when the loop is first entered.

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.

REVERSE

By default, iteration proceeds upward from the lower bound to the upper bound. However, if you use the keyword REVERSE, iteration proceeds downward from the upper bound to the lower bound.

An example follows:

```
FOR i IN REVERSE 1..10 LOOP  -- i starts at 10, ends at 1
   -- statements here execute 10 times
END LOOP;
```

The loop index is assigned the value of upper_bound. If that value is not less than the value of lower_bound, the sequence of statements in the loop is executed, then the index is decremented. If the value of the index is still not less than the value of lower_bound, the sequence of statements is executed again. This process repeats until the value of the index is less than the value of lower_bound. At that point, the loop completes.

cursor_for_loop_statement

A cursor FOR loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor, repeatedly fetches rows of values from the result set into fields in the record, and closes the cursor when all rows have been processed. Thus, the sequence of statements in the loop is executed once for each row that satisfies the query associated with cursor_name.

cursor_name

This identifies an explicit cursor previously declared within the current scope. When the cursor FOR loop is entered, cursor_name cannot refer to a cursor already opened by an OPEN statement or an enclosing cursor FOR loop.

record_name

This identifies an implicitly declared record. The record has the same structure as a row retrieved by cursor_name and is equivalent to a record declared as follows:

```
record_name cursor_name%ROWTYPE;
```

The record is defined only inside the loop. You cannot refer to its fields outside the loop. The implicit declaration of record_name overrides any other declaration outside the loop. So, another record with the same name cannot be referenced inside the loop unless a label is used.

Fields in the record store column values from the implicitly fetched row. The fields have the same names and datatypes as their corresponding columns. To access field values, you use dot notation, as follows:

```
record_name.field_name
```

Select-items fetched from the FOR loop cursor must have simple names or, if they are expressions, must have aliases. In the following example, wages is an alias for the select item sal+NVL(comm,0):

```
CURSOR c1 IS SELECT empno, sal+NVL(comm,0) wages, job ...
```

cursor_parameter_name

This identifies a cursor parameter; that is, a variable declared as the formal parameter of a cursor. (For the syntax of `cursor_parameter_declaration`, see "Cursors" on page 11-45.) A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be `IN` parameters.

select_statement

This is a query associated with an internal cursor unavailable to you. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 11-145. PL/SQL automatically declares, opens, fetches from, and closes the internal cursor. Because `select_statement` is not an independent statement, the implicit SQL cursor does not apply to it.

Usage Notes

You can use the `EXIT WHEN` statement to exit any loop prematurely. If the Boolean expression in the `WHEN` clause yields `TRUE`, the loop is exited immediately.

When you exit a cursor `FOR` loop, the cursor is closed automatically even if you use an `EXIT` or `GOTO` statement to exit the loop prematurely. The cursor is also closed automatically if an exception is raised inside the loop.

Example

The following cursor `FOR` loop calculates a bonus, then inserts the result into a database table:

```
DECLARE
   bonus  REAL;
   CURSOR c1 IS SELECT empno, sal, comm FROM emp;
BEGIN
   FOR c1rec IN c1 LOOP
      bonus := (c1rec.sal * 0.05) + (c1rec.comm * 0.25);
      INSERT INTO bonuses VALUES (c1rec.empno, bonus);
   END LOOP;
   COMMIT;
END;
```

Related Topics

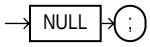
Cursors, EXIT Statement, FETCH Statement, OPEN Statement, %ROWTYPE Attribute

NULL Statement

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. In a construct allowing alternative actions, the NULL statement serves as a placeholder. For more information, see "NULL Statement" on page 3-19.

Syntax

null_statement



Usage Notes

The NULL statement improves readability by making the meaning and action of conditional statements clear. It tells readers that the associated alternative has not been overlooked, but that indeed no action is necessary.

Each clause in an IF statement must contain at least one executable statement. The NULL statement meets this requirement. So, you can use the NULL statement in clauses that correspond to circumstances in which no action is taken. The NULL statement and Boolean value NULL are unrelated.

Examples

In the following example, the NULL statement emphasizes that only salespeople receive commissions:

```
IF job_title = 'SALESPERSON' THEN
   compute_commission(emp_id);
ELSE
   NULL;
END IF;
```

In the next example, the NULL statement shows that no action is taken for unnamed exceptions:

```
EXCEPTION
   ...
   WHEN OTHERS THEN
      NULL;
```

Object Types

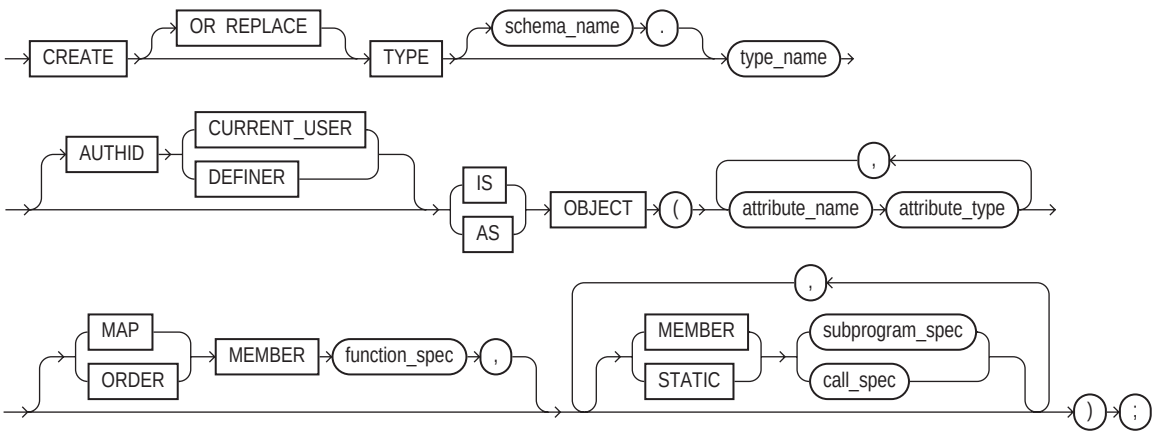
An object type is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called *attributes*. The functions and procedures that characterize the behavior of the object type are called *methods*.

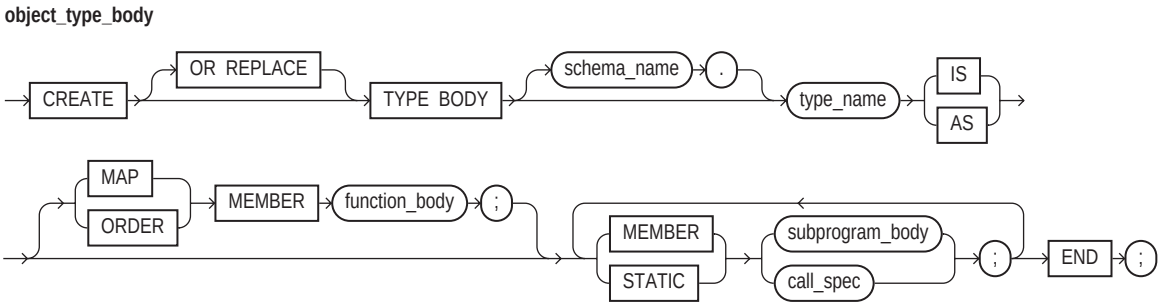
Currently, you cannot define object types within PL/SQL. They must be CREATEd and stored in an Oracle database, where they can be shared by many programs. When you define an object type (in SQL*Plus for example) using the CREATE TYPE statement, you create an abstract template for some real-world object. The template specifies only those attributes and behaviors the object will need in the application environment.

The data structure formed by the set of attributes is public (visible to client programs). However, well-behaved programs do not manipulate it directly. Instead, they use the set of methods provided. That way, the data is kept in a proper state. At run time, when the data structure is filled with values, you have created an *instance* of an object type. You can create as many instances (usually called *objects*) as you need. For more information, see Chapter 9, "Object Types".

Syntax

object_type_declaration | object_type_spec





Keyword and Parameter Description

type_name

This identifies a user-defined type specifier, which is used in subsequent declarations of objects.

AUTHID Clause

This determines whether all member methods execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights versus Definer Rights" on page 7-29.

attribute_name

This identifies an object attribute. The name must be unique within the object type (but can be reused in other object types). You cannot initialize an attribute in its declaration using the assignment operator or `DEFAULT` clause. Also, you cannot impose the `NOT NULL` constraint on an attribute.

attribute_datatype

This is any Oracle datatype except `LONG`, `LONG RAW`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `ROWID`, 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.

MEMBER | STATIC

This keyword allows you to declare a subprogram or call spec as a method in an object type spec. The method cannot have the same name as the object type or any of its attributes. MEMBER methods are invoked on instances, as in

```
instance_expression.method()
```

However, STATIC methods are invoked on the object type, not its instances, as in

```
object_type_name.method()
```

For each subprogram spec in an object type spec, there must be a corresponding subprogram body in the object type body. To match specs and bodies, the compiler does a token-by-token comparison of their headers. So, the headers must match word for word.

MEMBER methods accept a built-in parameter named SELF, which is an instance of the object type. Whether declared implicitly or explicitly, it is always the first parameter passed to a MEMBER method. However, STATIC methods cannot accept or reference SELF.

In the method body, SELF denotes the object whose method was invoked. For example, method transform declares SELF as an IN OUT parameter:

```
CREATE TYPE Complex AS OBJECT (
   MEMBER FUNCTION transform (SELF IN OUT Complex) ...
```

You cannot specify a different datatype for SELF. In MEMBER functions, if SELF is not declared, its parameter mode defaults to IN. However, in MEMBER procedures, if SELF is not declared, its parameter mode defaults to IN OUT. You cannot specify the OUT parameter mode for SELF.

MAP

This keyword indicates that a method orders objects by mapping them to values of a scalar datatype such as CHAR or REAL, which have a predefined order. PL/SQL uses the ordering to evaluate Boolean expressions such as x > y, and to do comparisons implied by the DISTINCT, GROUP BY, and ORDER BY clauses. A map method returns the relative position of an object in the ordering of all such objects.

An object type can contain only one map method, which must be a parameterless function having the return type DATE, NUMBER, VARCHAR2, or an ANSI SQL type such as CHARACTER, INTEGER, or REAL.

ORDER

This keyword indicates that a method compares two objects. An object type can contain only one order method, which must be a function that returns a numeric result.

Every order method takes just two parameters: the built-in parameter `SELF` and another object of the same type. If `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.

subprogram_spec

This construct declares the interface to a member function or procedure. Its syntax is like that of `function_spec` or `procedure_spec` without the terminator. See "Functions" on page 11-79 and / or "Procedures" on page 11-127.

subprogram_body

This construct defines the underlying implementation of a member function or procedure. Its syntax is like that of `function_body` or `procedure_body` without the terminator. See "Functions" on page 11-79 and / or "Procedures" on page 11-127.

call_spec

This publishes a Java method or external C function in the Oracle data dictionary. It publishes the routine by mapping its name, parameter types, and return type to their SQL counterparts. To learn how to write Java call specs, see *Oracle8i Java Stored Procedures Developer's Guide*. To learn how to write C call specs *Oracle8i Application Developer's Guide - Fundamentals*.

Usage Notes

Once an object type is defined and installed in the schema, you can use it to declare objects in any PL / SQL block, subprogram, or package. For example, you can use the object type to specify the datatype of an attribute, column, variable, bind variable, record field, table element, formal parameter, or function result.

Like a package, an object type has two parts: a specification and a body. The specification (spec for short) is the interface to your applications; it declares a data structure (set of attributes) along with the operations (methods) needed to manipulate the data. The body fully defines the methods, and so implements the spec.

All the information a client program needs to use the methods is in the spec. Think of the spec as an operational interface and of the body as a black box. You can debug, enhance, or replace the body without changing the spec.

An object type encapsulates data and operations. So, you can declare attributes and methods in an object type spec, but *not* constants, exceptions, cursors, or types. At least one attribute is required (the maximum is 1000); methods are optional.

In an object type spec, all attributes must be declared before any methods. Only subprograms have an underlying implementation. So, if an object type spec declares only attributes and/or call specs, the object type body is unnecessary. You cannot declare attributes in the body. All declarations in the object type spec are public (visible outside the object type).

You can refer to an attribute only by name (not by its position in the object type). To access or change the value of an attribute, you use dot notation. Attribute names can be chained, which allows you to access the attributes of a nested object type.

In an object type, methods can reference attributes and other methods without a qualifier. In SQL statements, calls to a parameterless method require an empty parameter list. In procedural statements, an empty parameter list is optional unless you chain calls, in which case it is required for all but the last call.

From a SQL statement, if you call a `MEMBER` method on a null instance (that is, `SELF` is null), the method is not invoked and a null is returned. From a procedural statement, if you call a `MEMBER` method on a null instance, PL/SQL raises the predefined exception `SELF_IS_NULL` before the method is invoked.

You can declare a map method or an order method but not both. If you declare either method, you can compare objects in SQL and procedural statements. However, if you declare neither method, you can compare objects only in SQL statements and only for equality or inequality. Two objects of the same type are equal *only if* the values of their corresponding attributes are equal.

Like packaged subprograms, methods of the same kind (functions or procedures) can be overloaded. That is, you can use the same name for different methods if their formal parameters differ in number, order, or datatype family.

Every object type has a constructor method (constructor for short), which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type. PL/SQL never calls a constructor implicitly, so you must call it explicitly. Constructor calls are allowed wherever function calls are allowed.

Examples

In the SQL*Plus script below, an object type for a stack is defined. The last item added to a stack is the first item removed. The operations *push* and *pop* update the stack while preserving last in, first out (LIFO) behavior. The simplest implementation of a stack uses an integer array. Integers are stored in array elements, with one end of the array representing the top of the stack.

```
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;
```

Notice that in member procedures `push` and `pop`, we use the built-in procedure `raise_application_error` to issue user-defined error messages. That way, we can report errors to the client program and avoid returning unhandled exceptions to the host environment.

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,
);
```

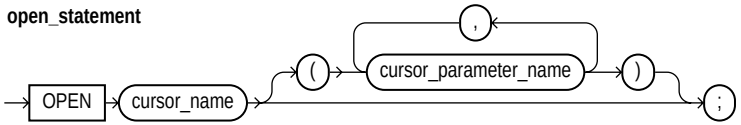
Related Topics

Functions, Packages, Procedures

OPEN Statement

The OPEN statement executes the multi-row query associated with an explicit cursor. It also allocates resources used by Oracle to process the query and identifies the result set, which consists of all rows that meet the query search criteria. The cursor is positioned before the first row in the result set. For more information, see "Managing Cursors" on page 5-6.

Syntax



Keyword and Parameter Description

cursor_name

This identifies an explicit cursor previously declared within the current scope and not currently open.

cursor_parameter_name

This identifies a cursor parameter; that is, a variable declared as the formal parameter of a cursor. (For the syntax of `cursor_parameter_declaration`, see "Cursors" on page 11-45.) A cursor parameter can appear in a query wherever a constant can appear.

Usage Notes

Generally, PL/SQL parses an explicit cursor only the first time it is opened and parses a SQL statement (thereby creating an implicit cursor) only the first time the statement is executed. All the parsed SQL statements are cached. A SQL statement must be reparsed only if it is aged out of the cache by a new SQL statement.

So, although you must close a cursor before you can reopen it, PL/SQL need not reparse the associated SELECT statement. If you close, then immediately reopen the cursor, a reparse is definitely not needed.

Rows in the result set are not retrieved when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows. With a FOR UPDATE cursor, the rows are locked when the cursor is opened.

If formal parameters are declared, actual parameters must be passed to the cursor. The formal parameters of a cursor must be IN parameters. Therefore, they cannot return values to actual parameters. The values of actual parameters are used when the cursor is opened. The datatypes of the formal and actual parameters must be compatible. The query can also reference PL/SQL variables declared within its scope.

Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. Formal parameters declared with a default value need not have a corresponding actual parameter. They can simply assume their default values when the OPEN statement is executed.

You can associate the actual parameters in an OPEN statement with the formal parameters in a cursor declaration using positional or named notation. For more information, see "Positional and Named Notation" on page 7-13.

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 CHAR, 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

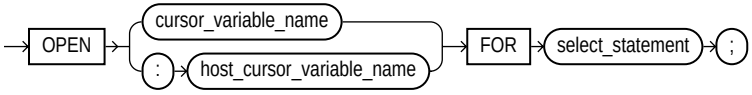
CLOSE Statement, Cursors, FETCH Statement, LOOP Statements

OPEN-FOR Statement

The OPEN-FOR statement executes the multi-row query associated with a cursor variable. It also allocates resources used by Oracle to process the query and identifies the result set, which consists of all rows that meet the query search criteria. The cursor variable is positioned before the first row in the result set. For more information, see "Using Cursor Variables" on page 5-15.

Syntax

open_for_statement



Keyword and Parameter Description

cursor_variable_name

This identifies a cursor variable (or parameter) previously declared within the current scope.

host_cursor_variable_name

This identifies a cursor variable previously declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

select_statement

This is a query associated with `cursor_variable`, which returns a set of values. The query can reference bind variables and PL/SQL variables, parameters, and functions but cannot be `FOR UPDATE`. The syntax of `select_statement` is similar to the syntax for `select_into_statement` defined in "SELECT INTO Statement" on page 11-145, except that `select_statement` cannot have an `INTO` clause.

Usage Notes

You can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program. To open the host cursor variable, you can pass it as a bind variable to an anonymous PL/SQL block. You can reduce network traffic by grouping OPEN-FOR statements. For example, the following PL/SQL block opens five cursor variables in a single round-trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
   OPEN :emp_cv FOR SELECT * FROM emp;
   OPEN :dept_cv FOR SELECT * FROM dept;
   OPEN :grade_cv FOR SELECT * FROM salgrade;
   OPEN :pay_cv FOR SELECT * FROM payroll;
   OPEN :ins_cv FOR SELECT * FROM insurance;
END;
```

Other OPEN-FOR statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. When you reopen a cursor variable for a different query, the previous query is lost.

Unlike cursors, cursor variables do not take parameters. No flexibility is lost, however, because you can pass whole queries (not just parameters) to a cursor variable.

You can pass a cursor variable to PL/SQL by calling a stored procedure that declares a cursor variable as one of its formal parameters. However, remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use a remote procedure call (RPC) to open a cursor variable.

When you declare a cursor variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the IN OUT mode. That way, the subprogram can pass an open cursor back to the caller.

Examples

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;
```

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 GenericCurTyp IS REF CURSOR;
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                          choice IN NUMBER);
END emp_data;

CREATE PACKAGE BODY emp_data AS
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                          choice IN NUMBER) IS
   BEGIN
      IF choice = 1 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE comm IS NOT NULL;
      ELSIF choice = 2 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE sal > 2500;
      ELSIF choice = 3 THEN
         OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = 20;
      END IF;
   END open_emp_cv;
END emp_data;
```

For more flexibility, you can pass a cursor variable to a stored procedure that executes queries with different return types, as follows:

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

Related Topics

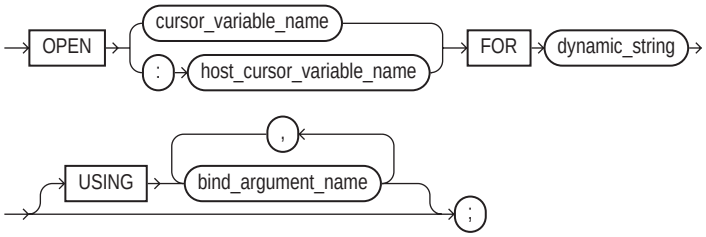
CLOSE Statement, Cursor Variables, FETCH Statement, LOOP Statements

OPEN-FOR-USING Statement

The OPEN-FOR-USING statement associates a cursor variable with a multi-row query, executes the query, identifies the result set, positions the cursor on the first row in the result set, then zeroes the rows-processed count kept by %ROWCOUNT. For more information, see Chapter 10, "Native Dynamic SQL".

Syntax

open_for_using_statement



Keyword and Parameter Description

dynamic_string

This is a string literal, variable, or expression that represents a multi-row SELECT statement.

cursor_variable_name

This identifies a a weakly typed cursor variable (one without a return type) previously declared within the current scope.

host_cursor_variable_name

This identifies a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

USING ...

This optional clause specifies a list of bind arguments. At run time, bind arguments in the `USING` clause replace corresponding placeholders in the dynamic `SELECT` statement.

bind_argument_name

This identifies an expression whose value is passed to 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 one at a time. When all the rows are processed, you `CLOSE` the cursor variable. (For more information about cursor variables, see "Using Cursor Variables" on page 5-15.)

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. For the right way, see "Passing the Names of Schema Objects" on page 10-10.

Every placeholder in the dynamic string must be associated with a bind argument in the `USING` clause. Numeric, character, and string literals are allowed in the `USING` clause, but Boolean literals (`TRUE`, `FALSE`, `NULL`) are not. To pass nulls to the dynamic string, you must use a workaround. See "Passing Nulls" on page 10-12.

Any bind arguments in the query are evaluated only when the cursor variable is opened. So, to fetch from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.

Dynamic SQL supports all the SQL datatypes. So, for example, bind arguments can be collections, `LOB`s, instances of an object type, and refs. As a rule, dynamic SQL does not support PL/SQL-specific types. So, for example, bind arguments cannot be Booleans or index-by tables.

Example

In the following example, we declare a cursor variable, then associate it with a dynamic SELECT statement that returns rows from the emp table:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;  -- define weak REF CURSOR type
   emp_cv   EmpCurTyp;  -- declare cursor variable
   my_ename VARCHAR2(15);
   my_sal   NUMBER := 1000;
BEGIN
   OPEN emp_cv FOR  -- open cursor variable
      'SELECT ename, sal FROM emp WHERE sal > :s' USING my_sal;
   ...
END;
```

Related Topics

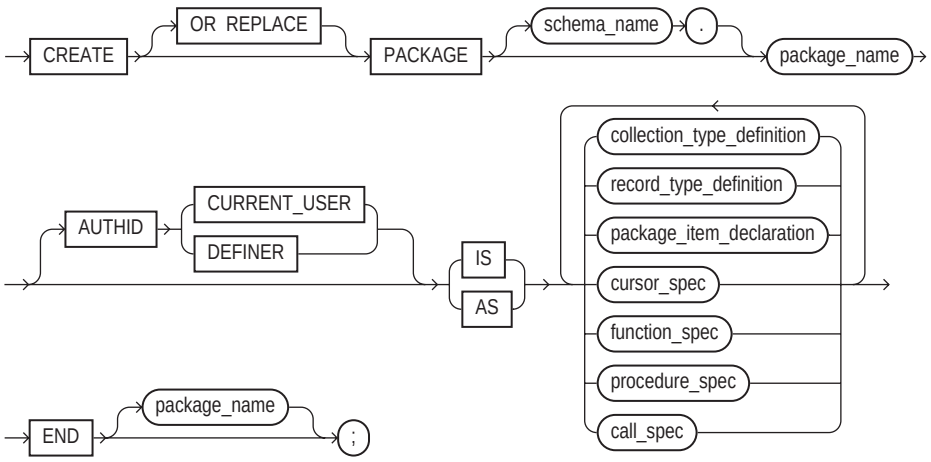
EXECUTE IMMEDIATE Statement

Packages

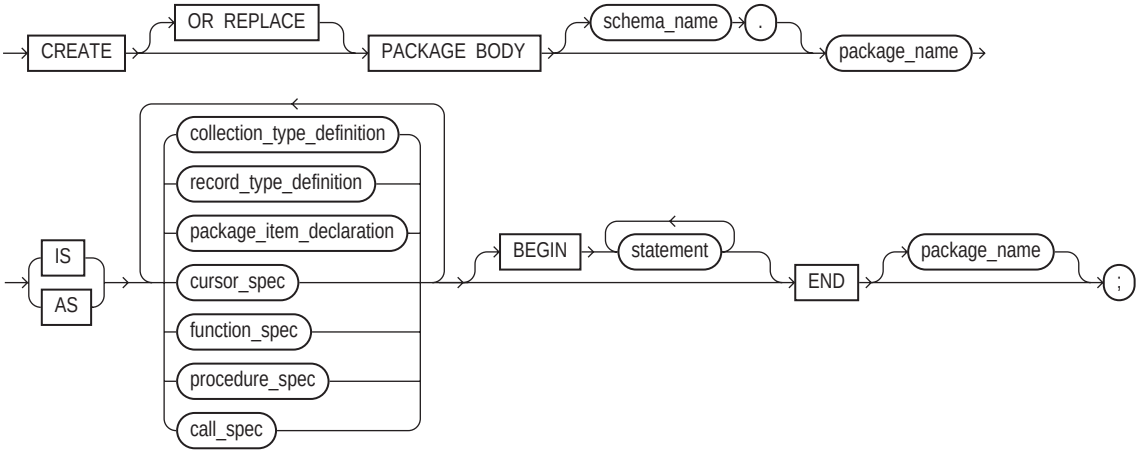
A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages have two parts: a specification (spec for short) and a body. For more information, see Chapter 8, "Packages".

Syntax

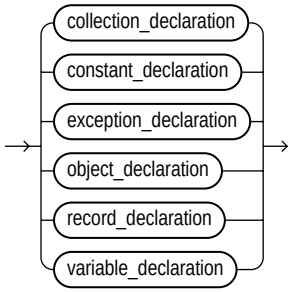
package_declaration | package_spec



package_body



package_item_declaration



Keyword and Parameter Description

package_name

This identifies a package. For naming conventions, see "Identifiers" on page 2-4.

AUTHID Clause

This determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker Rights versus Definer Rights" on page 7-29.

collection_declaration

This identifies a nested table, index-by table, or varray previously declared within the current scope. For the syntax of `collection_declaration`, see "Collections" on page 11-21.

constant_declaration

This construct declares a constant. For the syntax of `constant_declaration`, see "Constants and Variables" on page 11-30.

exception_declaration

This construct declares an exception. For the syntax of `exception_declaration`, see "Exceptions" on page 11-55.

object_declaration

This identifies an object (instance of an object type) previously declared within the current scope. For the syntax of `object_declaration`, see "Object Types" on page 11-105.

record_declaration

This construct declares a user-defined record. For the syntax of `record_declaration`, see "Records" on page 11-134.

variable_declaration

This construct declares a variable. For the syntax of `variable_declaration`, see "Constants and Variables" on page 11-30.

cursor_spec

This construct declares the interface to an explicit cursor. For the syntax of `cursor_spec`, see "Cursors" on page 11-45.

function_spec

This construct declares the interface to a function. For the syntax of `function_spec`, see "Functions" on page 11-79.

procedure_spec

This construct declares the interface to a procedure. For the syntax of `procedure_spec`, see "Procedures" on page 11-127.

call spec

This publishes a Java method or external C function in the Oracle data dictionary. It publishes the routine by mapping its name, parameter types, and return type to their SQL counterparts. For more information, see *Oracle8i Java Stored Procedures Developer's Guide* and / or *Oracle8i Application Developer's Guide - Fundamentals*.

cursor_body

This construct defines the underlying implementation of an explicit cursor. For the syntax of `cursor_body`, see "Cursors" on page 11-45.

function_body

This construct defines the underlying implementation of a function. For the syntax of `function_body`, see "Functions" on page 11-79.

procedure_body

This construct defines the underlying implementation of a procedure. For the syntax of `procedure_body`, see "Procedures" on page 11-127.

Usage Notes

You cannot define packages in a PL/SQL block or subprogram. However, you can use any Oracle tool that supports PL/SQL to create and store packages in an Oracle database. You can issue the CREATE PACKAGE and CREATE PACKAGE BODY statements from an Oracle Precompiler or OCI host program, or interactively from SQL*Plus.

Most packages have a spec and a body. The spec is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the spec.

Only subprograms and cursors have an underlying implementation (definition). So, if a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. However, the body can still be used to initialize items declared in the spec, as the following example shows:

```
CREATE PACKAGE emp_actions AS
   ...
   number_hired INTEGER;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
BEGIN
   number_hired := 0;
END emp_actions;
```

You can code and compile a spec without its body. Once the spec has been compiled, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application. Furthermore, you can debug, enhance, or replace a package body without changing the interface (package spec) to the package body. So, you need not recompile calling programs.

Cursors and subprograms declared in a package spec must be defined in the package body. Other program items declared in the package spec cannot be redeclared in the package body.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. So, except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception.

Related Topics

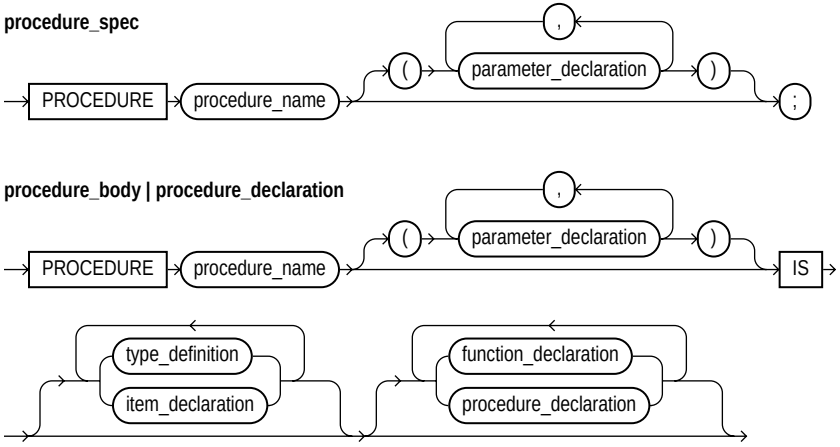
Collections, Cursors, Exceptions, Functions, Procedures, Records

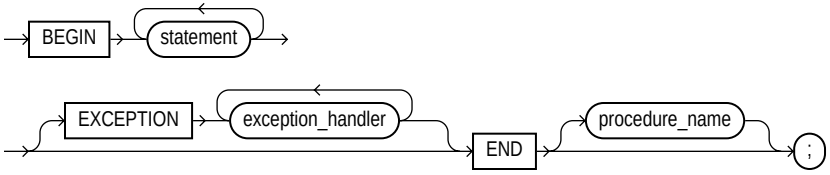
Procedures

A procedure is a subprogram that can take parameters and be invoked. Generally, you use a procedure to perform an action. A procedure has two parts: the specification and the body. The specification (spec for short) begins with the keyword PROCEDURE and ends with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses. The procedure body begins with the keyword IS and ends with the keyword END followed by an optional procedure name.

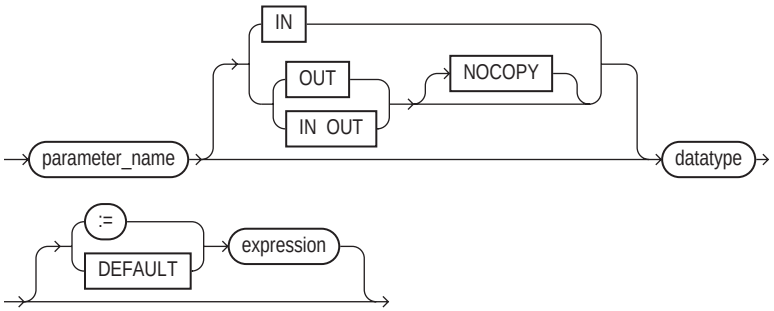
The procedure body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. These items are local and cease to exist when you exit the procedure. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution. For more information, see "Procedures" on page 7-3.

Syntax





parameter_declaration



Keyword and Parameter Description

procedure_name

This identifies a user-defined procedure.

parameter_name

This identifies a formal parameter, which is a variable declared in a procedure spec and referenced in the procedure body.

IN, OUT, IN OUT

These parameter modes define the behavior of formal parameters. An `IN` parameter lets you pass values to the subprogram being called. An `OUT` parameter lets you return values to the caller of the subprogram. An `IN OUT` parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

NOCOPY

This is a compiler hint (not directive), which allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference instead of by value (the default). For more information, see "NOCOPY Compiler Hint" on page 7-17.

datatype

This is a type specifier. For the syntax of `datatype`, see "Constants and Variables" on page 11-30.

:= | DEFAULT

This operator or keyword allows you to initialize `IN` parameters to default values.

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

type_definition

This specifies a user-defined datatype. For the syntax of `type_definition`, see "Blocks" on page 11-7.

item_declaration

This declares a program object. For the syntax of `item_declaration`, see "Blocks" on page 11-7.

function_declaration

This construct declares a function. For the syntax of `function_declaration`, see "Functions" on page 11-79.

procedure_declaration

This construct declares a procedure. For the syntax of `procedure_declaration`, see "Procedures" on page 11-127.

exception_handler

This construct 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 11-55.

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. So, you cannot assign it a value. An `OUT` parameter acts like a local variable. So, you can change its value and reference the value in any way. An `IN OUT` parameter acts like an initialized variable. So, you can assign it a value, which can be assigned to another variable. For summary information about the parameter modes, see Table 7–1 on page 7-16.

Unlike `OUT` and `IN OUT` parameters, `IN` parameters can be initialized to default values. For more information, see "Parameter Default Values" on page 7-19.

Before exiting a procedure, explicitly assign values to all `OUT` formal parameters. Otherwise, the values of corresponding actual parameters are indeterminate. If you exit successfully, PL/SQL assigns values to the actual parameters. However, if you exit with an unhandled exception, PL/SQL does *not* assign values to the actual parameters.

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.

Procedures can be defined using any Oracle tool that supports PL/SQL. To become available for general use, however, procedures must be `CREATE`d and stored in an Oracle database. You can issue the `CREATE PROCEDURE` statement interactively from SQL*Plus.

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;
```

In the following example, you call the procedure using named notation:

```
debit_account(amount => 500, acct_id => 10261);
```

Related Topics

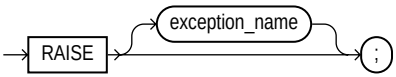
Collections, Functions, Packages, Records

RAISE Statement

The RAISE statement stops normal execution of a PL/SQL block or subprogram and transfers control to the appropriate exception handler. Normally, predefined exceptions are raised implicitly by the runtime system. However, RAISE statements can also raise predefined exceptions. User-defined exceptions must be raised explicitly by RAISE statements. For more information, see "User-Defined Exceptions" on page 6-7.

Syntax

raise_statement



Keyword and Parameter Description

exception_name

This identifies a predefined or user-defined exception. For a list of the predefined exceptions, see "Predefined Exceptions" on page 6-4.

Usage Notes

PL/SQL blocks and subprograms should RAISE an exception only when an error makes it undesirable or impossible to continue processing. You can code a RAISE statement for a given exception anywhere within the scope of that exception.

When an exception is raised, if PL/SQL cannot find a handler for it in the current block, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. In the latter case, PL/SQL returns an *unhandled exception* error to the host environment.

Omitting the exception name in a RAISE statement, which is allowed only in an exception handler, reraises the current exception. When a parameterless RAISE statement executes in an exception handler, the first block searched is the enclosing block, not the current block.

Example

In the following example, you raise an exception when an inventoried part is out of stock:

```
IF quantity_on_hand = 0 THEN
   RAISE out_of_stock;
END IF;
```

Related Topics

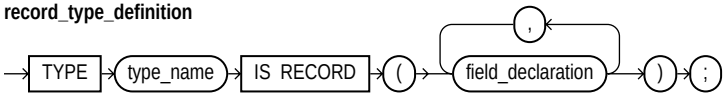
Exceptions

Records

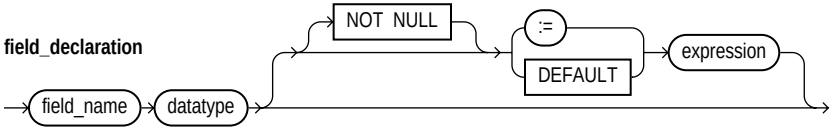
Records are items of type RECORD. Records have uniquely named fields that can store data values of different types. Thus, a record lets you treat related but dissimilar data as a logical unit. For more information, see "What Is a Record?" on page 4-37.

Syntax

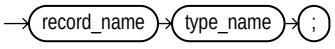
record_type_definition



field_declaration



record_declaration



Keyword and Parameter Description

record_type_name

This identifies a user-defined type specifier, which is used in subsequent declarations of records.

NOT NULL

This constraint prevents the assigning of nulls to a field. At run time, trying to assign a null to a field defined as NOT NULL raises the predefined exception VALUE_ERROR. The constraint NOT NULL must be followed by an initialization clause.

datatype

This is a type specifier. For the syntax of datatype, see "Constants and Variables" on page 11-30.

:= | DEFAULT

This operator or keyword allows you to initialize fields to default values.

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of `expression`, see "Expressions" on page 11-63. When the declaration is elaborated, the value of `expression` is assigned to the field. The value and the field must have compatible datatypes.

Usage Notes

You can define `RECORD` types and declare user-defined records in the declarative part of any block, subprogram, or package. Also, a record can be initialized in its declaration, as the following example shows:

```
DECLARE
   TYPE TimeTyp IS RECORD(
      seconds SMALLINT := 0,
      minutes SMALLINT := 0,
      hours   SMALLINT := 0);
```

The next example shows that you can use the `%TYPE` attribute to specify the datatype of a field. It also shows that you can add the `NOT NULL` constraint to any field declaration and thereby prevent the assigning of nulls to that field.

```
DECLARE
   TYPE DeptRecTyp IS RECORD(
      deptno NUMBER(2) NOT NULL,
      dname  dept.dname%TYPE,
      loc    dept.loc%TYPE);
   dept_rec DeptRecTyp;
```

To reference individual fields in a record, you use dot notation. For example, you might assign a value to the field `dname` in the record `dept_rec` as follows:

```
dept_rec.dname := 'PURCHASING';
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once. This can be done in two ways. First, you can assign one user-defined record to another if they have the same datatype. (Having fields that match exactly is not enough.) You can assign a `%ROWTYPE` record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes.

Second, you can use the SELECT or FETCH statement to fetch column values into a record. The columns in the select-list must appear in the same order as the fields in your record.

You can declare and reference nested records. That is, a record can be the component of another record, as the following example shows:

```
DECLARE
   TYPE TimeTyp IS RECORD(
      minutes SMALLINT,
      hours   SMALLINT);
   TYPE MeetingTyp IS RECORD(
      day     DATE,
      time_of TimeTyp,   -- nested record
      place   CHAR(20),
      purpose CHAR(50));
   TYPE PartyTyp IS RECORD(
      day     DATE,
      time_of TimeTyp,   -- nested record
      place   CHAR(15));
   meeting MeetingTyp;
   seminar MeetingTyp;
   party   PartyTyp;
```

The next example shows that 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 exit the application or end the database session. In a block or subprogram, they are instantiated when you enter the block or subprogram and cease to exist when you exit the block or subprogram.

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions. The restrictions that apply to scalar parameters also apply to user-defined records.

You can specify a RECORD type in the RETURN clause of a function spec. That allows the function to return a user-defined record of the same type. When calling a function that returns a user-defined record, use the following syntax to reference fields in the record:

```
function_name(parameter_list).field_name
```

To reference nested fields, use this syntax:

```
function_name(parameter_list).field_name.nested_field_name
```

If the function takes no parameters, code an empty parameter list. The syntax follows:

```
function_name().field_name
```

Example

In the following example, you define a RECORD type named DeptRecTyp, declare a record named dept_rec, then select a row of values into the record:

```
DECLARE
   TYPE DeptRecTyp IS RECORD(
      deptno NUMBER(2),
      dname  CHAR(14),
      loc    CHAR(13));
   dept_rec DeptRecTyp;
   ...
BEGIN
   SELECT deptno, dname, loc INTO dept_rec FROM dept
      WHERE deptno = 20;
```

Related Topics

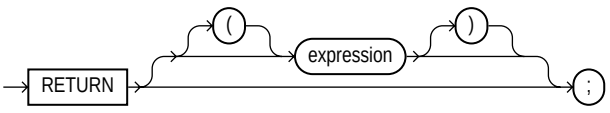
Assignment Statement, Collections, Functions, Procedures

RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. In a function, the RETURN statement also sets the function identifier to the result value. For more information, see "RETURN Statement" on page 7-8.

Syntax

return_statement



Keyword and Parameter Description

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the RETURN statement is executed, the value of `expression` is assigned to the function identifier.

Usage Notes

Do not confuse the RETURN statement with the RETURN clause, which specifies the datatype of the result value in a function spec.

A subprogram can contain several RETURN statements, none of which need be the last lexical statement. Executing any of them completes the subprogram immediately. However, to have multiple exit points in a subprogram is a poor programming practice.

In procedures, a RETURN statement cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a RETURN statement *must* contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier. Therefore, a function must contain at least one RETURN statement. Otherwise, PL/SQL raises the predefined exception PROGRAM_ERROR at run time.

The RETURN statement can also be used in an anonymous block to exit the block (and all enclosing blocks) immediately, but the RETURN statement cannot contain an expression.

Example

In the following example, the function balance returns the balance of a specified bank account:

```
FUNCTION balance (acct_id INTEGER) RETURN REAL IS
     acct_bal  REAL;
BEGIN
     SELECT bal INTO acct_bal FROM accts WHERE acctno = acct_id;
     RETURN acct_bal;
END balance;
```

Related Topics

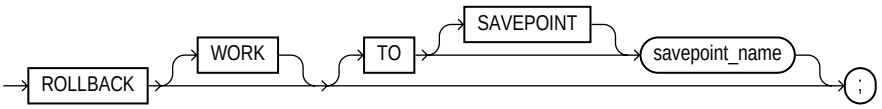
Functions, Procedures

ROLLBACK Statement

The ROLLBACK statement is the inverse of the COMMIT statement. It undoes some or all database changes made during the current transaction. For more information, see "Processing Transactions" on page 5-40.

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.

WORK

This keyword is optional and has no effect except to improve readability.

ROLLBACK TO

This statement undoes all database changes (and releases all locks acquired) since the savepoint identified by savepoint_name was marked.

SAVEPOINT

This keyword is optional and has no effect except to improve readability.

savepoint_name

This is an undeclared identifier, which marks the current point in the processing of a transaction. For naming conventions, see "Identifiers" on page 2-4.

Usage Notes

All savepoints marked after the savepoint to which you roll back are erased. However, the savepoint to which you roll back is not erased. For example, if you mark savepoints A, B, C, and D in that order, then roll back to savepoint B, only savepoints C and D are erased.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

In SQL, the FORCE clause manually rolls back an in-doubt distributed transaction. However, PL/SQL does not support this clause. For example, the following statement is illegal:

```
ROLLBACK WORK FORCE '24.37.85';  -- illegal
```

In embedded SQL, the RELEASE option frees all Oracle resources (locks and cursors) held by a program and disconnects from the database. However, PL/SQL does not support this option. For example, the following statement is illegal:

```
ROLLBACK WORK RELEASE;  -- illegal
```

Related Topics

COMMIT Statement, SAVEPOINT Statement

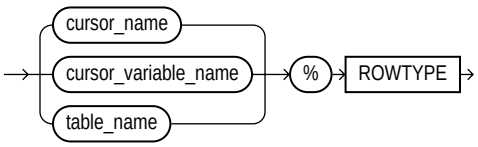
%ROWTYPE Attribute

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 %ROWTYPE" on page 2-32.

Syntax

rowtype_attribute



Keyword and Parameter Description

cursor_name

This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL strongly (not weakly) typed cursor variable previously declared within the current scope.

table_name

This identifies a database table (or view) that must be accessible when the declaration is elaborated.

Usage Notes

The %ROWTYPE attribute lets you declare records structured like a row of data in a database table. To reference a field in the record, you use dot notation. For example, you might reference the deptno field as follows:

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

You can assign the value of an expression to a specific field, as follows:

```
emp_rec.sal := average * 1.15;
```

There are two ways to assign values to all fields in a record at once. First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor. Second, you can assign a list of column values to a record by using the SELECT or FETCH statement. The column names must appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement. Select-items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases.

Examples

In the example below, you use %ROWTYPE to declare two records. The first record stores a row selected from the emp table. The second record stores a row fetched from the c1 cursor.

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

In the next example, you select a row from the emp table into a %ROWTYPE record:

```
DECLARE
   emp_rec  emp%ROWTYPE;
   ...
BEGIN
   SELECT * INTO emp_rec FROM emp WHERE empno = my_empno;
   IF (emp_rec.deptno = 20) AND (emp_rec.sal > 2000) THEN
      ...
   END IF;
END;
```

Related Topics

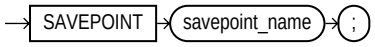
Constants and Variables, Cursors, Cursor Variables, FETCH Statement

SAVEPOINT Statement

The SAVEPOINT statement names and marks the current point in the processing of a transaction. With the ROLLBACK TO statement, savepoints let you undo parts of a transaction instead of the whole transaction. For more information, see "Processing Transactions" on page 5-40.

Syntax

savepoint_statement



Keyword and Parameter Description

savepoint_name

This is an undeclared identifier, which marks the current point in the processing of a transaction.

Usage Notes

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. However, the savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints. Savepoint names can be reused within a transaction. This moves the savepoint from its old position to the current point in the transaction.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent. However, you can only roll back to the most recently marked savepoint.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

Related Topics

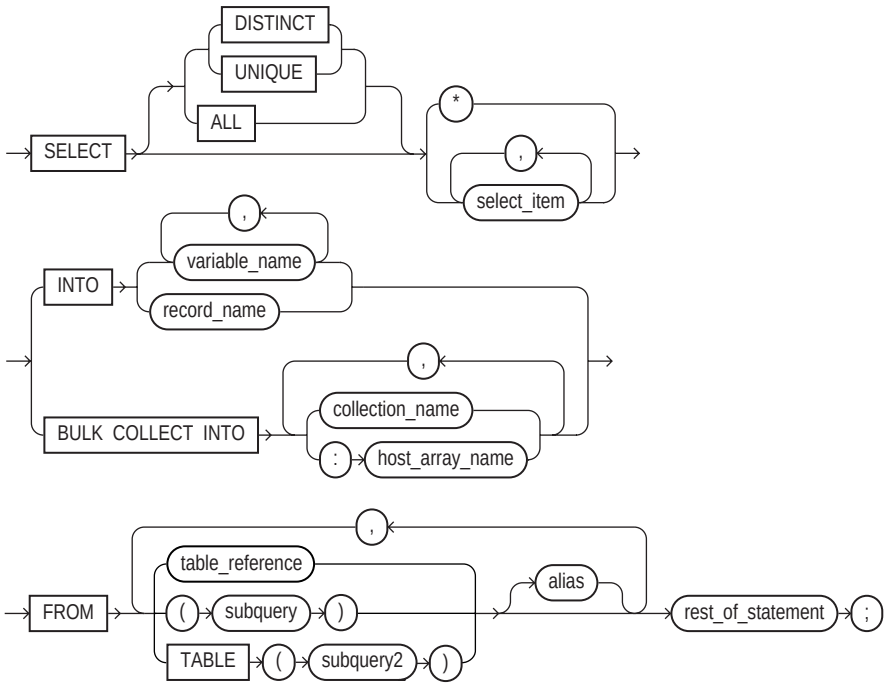
COMMIT Statement, ROLLBACK Statement

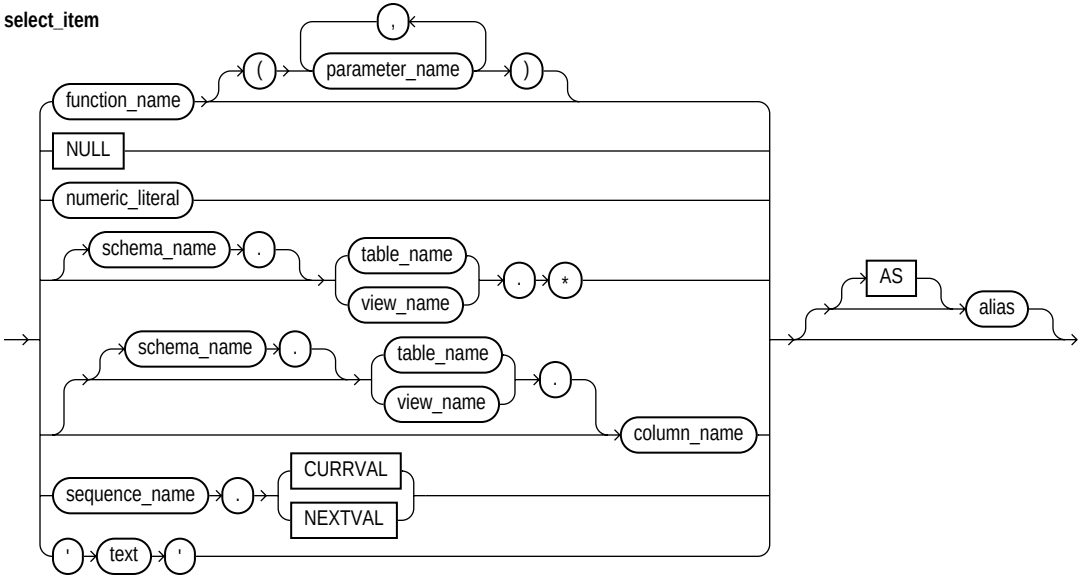
SELECT INTO Statement

The SELECT INTO statement retrieves data from one or more database tables, then assigns the selected values to variables or fields. For a full description of the SELECT statement, see *Oracle8i SQL Reference*.

Syntax

select_into_statement





Keyword and Parameter Description

select_item

This is a value returned by the `SELECT` statement, then assigned to the corresponding variable or field in the `INTO` clause.

BULK COLLECT

This clause instructs the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. The SQL engine bulk-binds all collections referenced in the `INTO` list. The corresponding columns must store scalar (not composite) values. For more information, see "Taking Advantage of Bulk Binds" on page 4-29.

variable_name

This identifies a previously declared scalar 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.

record_name

This identifies a user-defined or %ROWTYPE record into which rows of values are fetched. For each select_item value returned by the query, there must be a corresponding, type-compatible field in the record.

collection_name

This identifies a declared collection into which select_item values are bulk fetched. For each select_item, there must be a corresponding, type-compatible collection in the list.

host_array_name

This identifies an array (declared in a PL/SQL host environment and passed to PL/SQL as a bind variable) into which select_item values are bulk fetched. For each select_item, there must be a corresponding, type-compatible array in the list. Host arrays must be prefixed with a colon.

table_reference

This identifies a table or view that must be accessible when you execute the SELECT statement, and for which you must have SELECT privileges. For the syntax of table_reference, see "DELETE Statement" on page 11-49.

subquery

This is a SELECT statement that provides a set of rows for processing. Its syntax is like that of select_into_statement without the INTO clause. See "SELECT INTO Statement" on page 11-145.

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 cast as a nested table. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

alias

This is another (usually short) name for the referenced column, table, or view.

rest_of_statement

This is anything that can legally follow the FROM clause in a SELECT statement except the SAMPLE clause.

Usage Notes

The implicit SQL cursor and the cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of a SELECT INTO statement.

When you use a SELECT INTO statement to assign values to variables, it should return only one row. If it returns more than one row, you get the following results:

- PL/SQL raises the predefined exception TOO_MANY_ROWS
- SQLCODE returns -1422 (Oracle error code ORA-01422)
- SQLERRM returns the Oracle error message *single-row query returns more than one row*
- SQL%NOTFOUND yields FALSE
- SQL%FOUND yields TRUE
- SQL%ROWCOUNT yields 1

If no rows are returned, you get these results:

- PL/SQL raises the predefined exception NO_DATA_FOUND unless the SELECT statement called a SQL aggregate function such as AVG or SUM. (SQL aggregate functions always return a value or a null. So, a SELECT INTO statement that calls a aggregate function never raises NO_DATA_FOUND.)
- SQLCODE returns +100 (Oracle error code ORA-01403)
- SQLERRM returns the Oracle error message *no data found*
- SQL%NOTFOUND yields TRUE
- SQL%FOUND yields FALSE
- SQL%ROWCOUNT yields 0

Example

The following SELECT statement returns an employee's name, job title, and salary from the emp database table:

```
SELECT ename, job, sal INTO my_ename, my_job, my_sal FROM emp
   WHERE empno = my_empno;
```

Related Topics

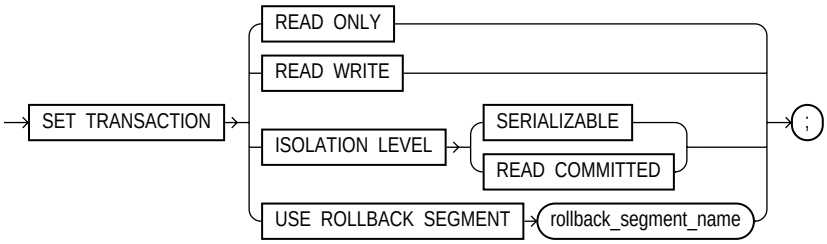
Assignment Statement, FETCH Statement, %ROWTYPE Attribute

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 "Using SET TRANSACTION" on page 5-46.

Syntax

set_transaction_statement



Keyword and Parameter Description

READ ONLY

This clause establishes the current transaction as read-only. If a transaction is set to READ ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

READ WRITE

This clause establishes the current transaction as read-write. The use of READ WRITE does not affect other users or transactions. If the transaction executes a data manipulation statement, Oracle assigns the transaction to a rollback segment.

ISOLATION LEVEL

This clause specifies how transactions that modify the database are handled. When you specify SERIALIZABLE, if a serializable transaction tries to execute a SQL data manipulation statement that modifies any table already modified by an uncommitted transaction, the statement fails.

To enable SERIALIZABLE mode, your DBA must set the Oracle initialization parameter COMPATIBLE to 7.3.0 or higher.

When you specify READ COMMITTED, if a transaction includes SQL data manipulation statements that require row locks held by another transaction, the statement waits until the row locks are released.

USE ROLLBACK SEGMENT

This clause assigns the current transaction to the specified rollback segment and establishes the transaction as read-write. You cannot use this parameter with the READ ONLY parameter in the same transaction because read-only transactions do not generate rollback information.

Usage Notes

The SET TRANSACTION statement must be the first SQL statement in your transaction and can appear only once in the transaction.

Example

In the following example, you establish a read-only transaction:

```
COMMIT;  -- end previous transaction
SET TRANSACTION READ ONLY;
SELECT ... FROM emp WHERE ...
SELECT ... FROM dept WHERE ...
SELECT ... FROM emp WHERE ...
COMMIT;  -- end read-only transaction
```

Related Topics

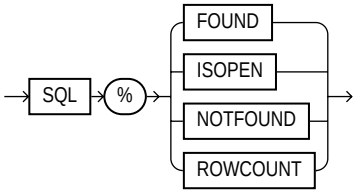
COMMIT Statement, ROLLBACK Statement, SAVEPOINT Statement

SQL Cursor

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicit cursor. PL/SQL lets you refer to the most recent implicit cursor as the SQL cursor, which has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. They give you useful information about the execution of data manipulation statements. For more information, see "Managing Cursors" on page 5-6.

Syntax

sql_cursor



Keyword and Parameter Description

SQL

This is the name of the implicit SQL cursor.

%FOUND

This attribute yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it yields FALSE.

%ISOPEN

This attribute always yields FALSE because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%NOTFOUND

This attribute is the logical opposite of %FOUND. It yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it yields FALSE.

%ROWCOUNT

This attribute yields the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Usage Notes

You can use cursor attributes in procedural statements but not in SQL statements. Before Oracle opens the SQL cursor automatically, the implicit cursor attributes yield NULL.

The values of cursor attributes always refer to the most recently executed SQL statement, wherever that statement appears. It might be in a different scope. So, if you want to save an attribute value for later use, assign it to a Boolean variable immediately.

If a SELECT INTO statement fails to return a row, PL/SQL raises the predefined exception NO_DATA_FOUND whether you check SQL%NOTFOUND on the next line or not. However, a SELECT INTO statement that calls a SQL aggregate function never raises NO_DATA_FOUND. That is because aggregate functions such as AVG and SUM always return a value or a null. In such cases, SQL%NOTFOUND yields FALSE.

Examples

In the following example, %NOTFOUND is used to insert a row if an update affects no rows:

```
UPDATE emp SET sal = sal * 1.05 WHERE empno = my_empno;
IF SQL%NOTFOUND THEN
   INSERT INTO emp VALUES (my_empno, my_ename, ...);
END IF;
```

In the next example, you use %ROWCOUNT to raise an exception if more than 100 rows are deleted:

```
DELETE FROM parts WHERE status = 'OBSOLETE';
IF SQL%ROWCOUNT > 100 THEN  -- more than 100 rows were deleted
   RAISE large_deletion;
END IF;
```

Related Topics

Cursors, Cursor Attributes

SQLCODE Function

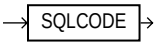
The function SQLCODE returns the number code associated with the most recently raised exception. SQLCODE is meaningful only in an exception handler. Outside a handler, SQLCODE always returns 0.

For internal exceptions, SQLCODE returns the number of the associated Oracle error. The number that SQLCODE returns is negative unless the Oracle error is *no data found*, in which case SQLCODE returns +100.

For user-defined exceptions, SQLCODE returns +1 unless you used the pragma EXCEPTION_INIT to associate the exception with an Oracle error number, in which case SQLCODE returns that error number. For more information, see "Using SQLCODE and SQLERRM" on page 6-18.

Syntax

sqlcode_function



Usage Notes

You cannot use SQLCODE directly in a SQL statement. First, you must assign the value of SQLCODE to a local variable. An example follows:

```
DECLARE
   my_sqlcode  NUMBER;
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN
      my_sqlcode := SQLCODE;
      INSERT INTO errors VALUES (my_sqlcode, ...);
END;
```

SQLCODE is especially useful in the OTHERS exception handler because it lets you identify which internal exception was raised.

When using pragma RESTRICT_REFERENCES to assert the purity of a packaged function, you cannot specify the restrictions WNPS and RNPS if the function calls SQLCODE.

Related Topics

Exceptions, SQLERRM Function

SQLERRM Function

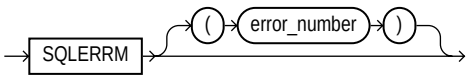
The function SQLERRM returns the error message associated with its error-number argument or, if the argument is omitted, with the current value of SQLCODE. SQLERRM with no argument is meaningful only in an exception handler. Outside a handler, SQLERRM with no argument always returns the message *normal, successful completion*.

For internal exceptions, SQLERRM returns the message associated with the Oracle error that occurred. The message begins with the Oracle error code.

For user-defined exceptions, SQLERRM returns the message *user-defined exception* unless you used the pragma EXCEPTION_INIT to associate the exception with an Oracle error number, in which case SQLERRM returns the corresponding error message. For more information, see "Using SQLCODE and SQLERRM" on page 6-18.

Syntax

sqlerrm_function



Keyword and Parameter Description

error_number

This must be a valid Oracle error number. For a list of Oracle errors, see *Oracle8i Error Messages*.

Usage Notes

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. The error number passed to SQLERRM should be negative. Passing a zero to SQLERRM always returns the following message:

```
ORA-0000: normal, successful completion
```

Passing a positive number to SQLERRM always returns the message

```
User-Defined Exception
```

unless you pass +100, in which case SQLERRM returns the following message:

```
ORA-01403: no data found
```

You cannot use SQLERRM directly in a SQL statement. First, you must assign the value of SQLERRM to a local variable. An example follows:

```
DECLARE
   my_sqlerrm  CHAR(150);
   ...
BEGIN
   ...
EXCEPTION
   ...
   WHEN OTHERS THEN
   my_sqlerrm := SUBSTR(SQLERRM, 1, 150);
   INSERT INTO errors VALUES (my_sqlerrm, ...);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to my_sqlerrm. SQLERRM is especially useful in the OTHERS exception handler because it lets you identify which internal exception was raised.

When using pragma RESTRICT_REFERENCES to assert the purity of a packaged function, you cannot specify the restrictions WNPS and RNPS if the function calls SQLERRM.

Related Topics

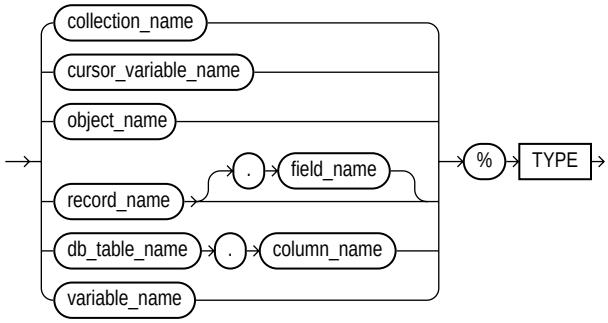
Exceptions, SQLCODE Function

%TYPE Attribute

The %TYPE attribute provides the datatype of a field, record, nested table, database column, or variable. You can use the %TYPE attribute as a datatype specifier when declaring constants, variables, fields, and parameters. For more information, see "Using %TYPE" on page 2-31.

Syntax

type_attribute



Keyword and Parameter Description

collection_name

This identifies a nested table, index-by table, or varray previously declared within the current scope.

cursor_variable_name

This identifies a PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

object_name

This identifies an object (instance of an object type) previously declared within the current scope.

record_name

This identifies a user-defined or %ROWTYPE record previously declared within the current scope.

record_name.field_name

This identifies a field in a user-defined or %ROWTYPE record previously declared within the current scope.

table_name.column_name

This refers to a table and column that must be accessible when the declaration is elaborated.

variable_name

This identifies a variable previously declared in the same scope.

Usage Notes

The %TYPE attribute is particularly useful when declaring variables, fields, and parameters that refer to database columns. However, the NOT NULL column constraint is *not* inherited by items declared using %TYPE.

Related Topics

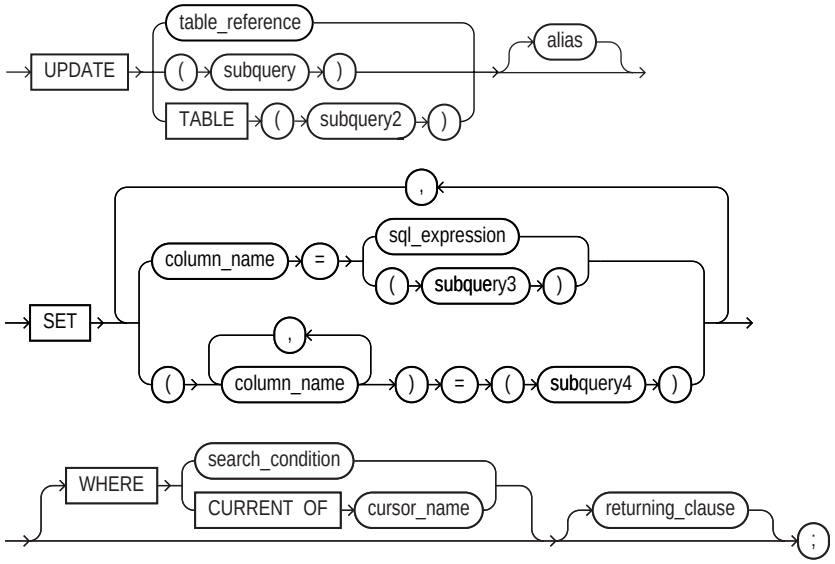
Constants and Variables, %ROWTYPE Attribute

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 *Oracle8i SQL Reference*.

Syntax

update_statement



Keyword and Parameter Description

table_reference

This identifies a table or view that must be accessible when you execute the UPDATE statement, and for which you must have UPDATE privileges. For the syntax of `table_reference`, see "DELETE Statement" on page 11-49.

subquery

This is a SELECT statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "SELECT INTO Statement" on page 11-145.

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 cast as a nested table. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

alias

This is another (usually short) name for the referenced table or view and is typically used in the WHERE clause.

column_name

This is the name of the column (or one of the columns) to be updated. It must be the name of a column in the referenced table or view. A column name cannot be repeated in the `column_name` list. Column names need not appear in the UPDATE statement in the same order that they appear in the table or view.

sql_expression

This is any valid SQL expression. For more information, see *Oracle8i SQL Reference*.

SET column_name = sql_expression

This clause assigns the value of `sql_expression` to the column identified by `column_name`. If `sql_expression` contains references to columns in the table being updated, the references are resolved in the context of the current row. The old column values are used on the right side of the equal sign.

In the following example, you increase every employee's salary by 10%. The original value of the `sal` column is multiplied by `1.10`, then the result is assigned to the `sal` column overwriting the original value.

```
UPDATE emp SET sal = sal * 1.10;
```

SET column_name = (subquery3)

This clause assigns the value retrieved from the database by `subquery3` to the column identified by `column_name`. The subquery must return exactly one row and one column.

SET (column_name, column_name, ...) = (subquery4)

This clause assigns the values retrieved from the database by `subquery4` to the columns in the `column_name` list. The subquery must return exactly one row that includes all the columns listed.

The column values returned by the subquery are assigned to the columns in the column list in order. The first value is assigned to the first column in the list, the second value is assigned to the second column in the list, and so on.

In the following correlated query, the column `item_id` is assigned the value stored in `item_num`, and the column `price` is assigned the value stored in `item_price`:

```
UPDATE inventory inv  -- alias
   SET (item_id, price) =
      (SELECT item_num, item_price FROM item_table
         WHERE item_name = inv.item_name);
```

WHERE search_condition

This clause chooses which rows to update in the database table. Only rows that meet the search condition are updated. If you omit the search condition, all rows in the table are updated.

WHERE CURRENT OF cursor_name

This clause refers to the latest row processed by the `FETCH` statement associated with the cursor identified by `cursor_name`. The cursor must be `FOR UPDATE` and must be open and positioned on a row.

If the cursor is not open, the `CURRENT OF` clause causes an error. If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception `NO_DATA_FOUND`.

returning_clause

This clause lets you return values from updated rows, thereby eliminating the need to `SELECT` the rows afterward. You can retrieve the column values into variables and/or host variables, or into collections and/or host arrays. However, you cannot use the `RETURNING` clause for remote or parallel updates. For the syntax of `returning_clause`, see "DELETE Statement" on page 11-49.

Usage Notes

You can use the UPDATE WHERE CURRENT OF statement after a fetch from an open cursor (this includes implicit fetches executed in a cursor FOR loop), provided the associated query is FOR UPDATE. This statement updates the current row; that is, the one just fetched.

The implicit SQL cursor and the cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN let you access useful information about the execution of an UPDATE statement.

An UPDATE statement might update one or more rows or no rows. If one or more rows are updated, you get the following results:

- SQL%NOTFOUND yields FALSE
- SQL%FOUND yields TRUE
- SQL%ROWCOUNT yields the number of rows updated

If no rows are updated, you get these results:

- SQL%NOTFOUND yields TRUE
- SQL%FOUND yields FALSE
- SQL%ROWCOUNT yields 0

Examples

In the following example, a 10% raise is given to analysts in department 20:

```
UPDATE emp SET sal = sal * 1.10
   WHERE job = 'ANALYST' AND DEPTNO = 20;
```

In the next example, an employee named Ford is promoted to the position of Analyst and her salary is raised by 15%:

```
UPDATE emp SET job = 'ANALYST', sal = sal * 1.15
   WHERE ename = 'FORD';
```

In the final example, values returned from an updated row are stored in variables:

```
UPDATE emp SET sal = sal + 500 WHERE ename = 'MILLER'
   RETURNING sal, ename INTO my_sal, my_ename;
```

Related Topics

DELETE Statement, FETCH Statement

A

Sample Programs

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

Major Topics

Running the Programs

All the sample programs in this appendix and several others throughout this guide are available online. So, they are preceded by the following comment:

```
-- available online in file '<filename>'
```

You can find the online files in the PL/SQL demo directory. For the location of the directory, see the Oracle installation or user's guide for your system. Here is a list of the files and their locations in this guide:

Filename	Location in This Guide
examp1	"Main Features" on page 1-2
examp2	"Conditional Control" on page 1-8
examp3	"Iterative Control" on page 1-9
examp4	"Using Aliases" on page 2-34
examp7	"Using Cursor FOR Loops" on page 5-13
examp8	"Passing Parameters" on page 5-15
examp5	"Some Examples" on page 5-36
examp6	"Some Examples" on page 5-36
examp11	"Example" on page 11-12
examp12	"Examples" on page 11-36
examp13	"Examples" on page 11-36
examp14	"Examples" on page 11-36
sample1	"Sample 1. FOR Loop" on page A-3
sample2	"Sample 2. Cursors" on page A-4
sample3	"Sample 3. Scoping" on page A-6
sample4	"Sample 4. Batch Transaction Processing" on page A-7
sample5	"Sample 5. Embedded PL/SQL" on page A-11
sample6	"Sample 6. Calling a Stored Procedure" on page A-15

You run some samples interactively from SQL*Plus, others from Pro*C programs. You can experiment with the samples from any Oracle account. However, the Pro*C examples expect you to use the `scott/tiger` account.

Before trying the samples, you must create some database tables, then load the tables with data. You do that by running two SQL*Plus scripts, `exampbld` and `examplod`, which are supplied with PL/SQL. You can find these scripts in the PL/SQL demo directory.

The first script builds the database tables processed by the sample programs. The second script loads (or reloads) the database tables. To run the scripts, invoke SQL*Plus, then issue the following commands:

```
SQL> START exampbld
...
SQL> START examplod
```

Sample 1. FOR Loop

The following example uses a simple `FOR` loop to insert ten rows into a database table. The values of a loop index, counter variable, and either of two character strings are inserted. Which string is inserted depends on the value of the loop index.

Input Table

Not applicable.

PL/SQL Block

```
-- available online in file 'sample1'
DECLARE
   x NUMBER := 100;
BEGIN
   FOR i IN 1..10 LOOP
      IF MOD(i,2) = 0 THEN     -- i is even
         INSERT INTO temp VALUES (i, x, 'i is even');
      ELSE
         INSERT INTO temp VALUES (i, x, 'i is odd');
      END IF;
      x := x + 100;
   END LOOP;
   COMMIT;
END;
```

Output Table

```
SQL> SELECT * FROM temp ORDER BY col1;

 COL1    COL2    MESSAGE
----- -------    ---------
    1     100    i is odd
    2     200    i is even
    3     300    i is odd
    4     400    i is even
    5     500    i is odd
    6     600    i is even
    7     700    i is odd
    8     800    i is even
    9     900    i is odd
   10    1000    i is even
```

Sample 2. Cursors

The following example uses a cursor to select the five highest paid employees from the emp table.

Input Table

```
SQL> SELECT ename, empno, sal FROM emp ORDER BY sal DESC;

ENAME             EMPNO      SAL
---------- ----------- --------
KING               7839     5000
SCOTT              7788     3000
FORD               7902     3000
JONES              7566     2975
BLAKE              7698     2850
CLARK              7782     2450
ALLEN              7499     1600
TURNER             7844     1500
MILLER             7934     1300
WARD               7521     1250
MARTIN             7654     1250
ADAMS              7876     1100
JAMES              7900      950
SMITH              7369      800
```

PL/SQL Block

```
-- available online in file 'sample2'
DECLARE
   CURSOR c1 is
      SELECT ename, empno, sal FROM emp
         ORDER BY sal DESC;   -- start with highest paid employee
   my_ename CHAR(10);
   my_empno NUMBER(4);
   my_sal   NUMBER(7,2);
BEGIN
   OPEN c1;
   FOR i IN 1..5 LOOP
      FETCH c1 INTO my_ename, my_empno, my_sal;
      EXIT WHEN c1%NOTFOUND;  /* in case the number requested */
                              /* is more than the total       */
                              /* number of employees          */
      INSERT INTO temp VALUES (my_sal, my_empno, my_ename);
      COMMIT;
   END LOOP;
   CLOSE c1;
END;
```

Output Table

```
SQL> SELECT * FROM temp ORDER BY col1 DESC;

     COL1     COL2    MESSAGE
--------- --------    -------
     5000     7839    KING
     3000     7902    FORD
     3000     7788    SCOTT
     2975     7566    JONES
     2850     7698    BLAKE
```

Sample 3. Scoping

The following example illustrates block structure and scope rules. An outer block declares two variables named `x` and `counter` and loops four times. Inside this loop is a sub-block that also declares a variable named `x`. The values inserted into the `temp` table show that the two `x`'s are indeed different.

Input Table

Not applicable.

PL/SQL Block

```
-- available online in file 'sample3'
DECLARE
   x NUMBER := 0;
   counter NUMBER := 0;
BEGIN
   FOR i IN 1..4 LOOP
      x := x + 1000;
      counter := counter + 1;
      INSERT INTO temp VALUES (x, counter, 'outer loop');
      /* start an inner block */
      DECLARE
         x NUMBER := 0;  -- this is a local version of x
      BEGIN
         FOR i IN 1..4 LOOP
            x := x + 1;  -- this increments the local x
            counter := counter + 1;
            INSERT INTO temp VALUES (x, counter, 'inner loop');
         END LOOP;
      END;
   END LOOP;
   COMMIT;
END;
```

Output Table

```
SQL> SELECT * FROM temp ORDER BY col2;

   COL1          COL2    MESSAGE
-------      --------    -------------
   1000             1    OUTER loop
      1             2    inner loop
      2             3    inner loop
      3             4    inner loop
      4             5    inner loop
   2000             6    OUTER loop
      1             7    inner loop
      2             8    inner loop
      3             9    inner loop
      4            10    inner loop
   3000            11    OUTER loop
      1            12    inner loop
      2            13    inner loop
      3            14    inner loop
      4            15    inner loop
   4000            16    OUTER loop
      1            17    inner loop
      2            18    inner loop
      3            19    inner loop
      4            20    inner loop
```

Sample 4. Batch Transaction Processing

In the next example the `accounts` table is modified according to instructions stored in the `action` table. Each row in the `action` table contains an account number, an action to be taken (I, U, or D for insert, update, or delete), an amount by which to update the account, and a time tag used to sequence the transactions.

On an insert, if the account already exists, an update is done instead. On an update, if the account does not exist, it is created by an insert. On a delete, if the row does not exist, no action is taken.

Input Tables

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID         BAL
----------   ---------
         1        1000
         2        2000
         3        1500
         4        6500
         5         500

SQL> SELECT * FROM action ORDER BY time_tag;

ACCOUNT_ID  O  NEW_VALUE STATUS                    TIME_TAG
----------  - ---------- ------------------------ ---------
         3  u        599                          18-NOV-88
         6  i      20099                          18-NOV-88
         5  d                                     18-NOV-88
         7  u       1599                          18-NOV-88
         1  i        399                          18-NOV-88
         9  d                                     18-NOV-88
        10  x                                     18-NOV-88
```

PL/SQL Block

```
-- available online in file 'sample4'
DECLARE
   CURSOR c1 IS
      SELECT account_id, oper_type, new_value FROM action
      ORDER BY time_tag
      FOR UPDATE OF status;
BEGIN
   FOR acct IN c1 LOOP  -- process each row one at a time

   acct.oper_type := upper(acct.oper_type);

   /*--------------------------------------*/
   /* Process an UPDATE.  If the account to  */
   /* be updated doesn't exist, create a new */
   /* account.                               */
   /*--------------------------------------*/
   IF acct.oper_type = 'U' THEN
      UPDATE accounts SET bal = acct.new_value
         WHERE account_id = acct.account_id;
```

```
   IF SQL%NOTFOUND THEN  -- account didn't exist. Create it.
      INSERT INTO accounts
         VALUES (acct.account_id, acct.new_value);
      UPDATE action SET status =
         'Update: ID not found. Value inserted.'
         WHERE CURRENT OF c1;
   ELSE
      UPDATE action SET status = 'Update: Success.'
         WHERE CURRENT OF c1;
   END IF;

/*-------------------------------------------*/
/* Process an INSERT.  If the account already */
/* exists, do an update of the account       */
/* instead.                                  */
/*-------------------------------------------*/
ELSIF acct.oper_type = 'I' THEN
   BEGIN
      INSERT INTO accounts
         VALUES (acct.account_id, acct.new_value);
      UPDATE action set status = 'Insert: Success.'
         WHERE CURRENT OF c1;
      EXCEPTION
         WHEN DUP_VAL_ON_INDEX THEN   -- account already exists
            UPDATE accounts SET bal = acct.new_value
               WHERE account_id = acct.account_id;
            UPDATE action SET status =
               'Insert: Acct exists. Updated instead.'
               WHERE CURRENT OF c1;
    END;

/*-------------------------------------------*/
/* Process a DELETE.  If the account doesn't  */
/* exist, set the status field to say that    */
/* the account wasn't found.                 */
/*-------------------------------------------*/
ELSIF acct.oper_type = 'D' THEN
   DELETE FROM accounts
      WHERE account_id = acct.account_id;

   IF SQL%NOTFOUND THEN   -- account didn't exist.
      UPDATE action SET status = 'Delete: ID not found.'
         WHERE CURRENT OF c1;
   ELSE
```

```
         UPDATE action SET status = 'Delete: Success.'
            WHERE CURRENT OF c1;
      END IF;

   /*-------------------------------------------*/
   /* The requested operation is invalid.       */
   /*-------------------------------------------*/
   ELSE  -- oper_type is invalid
      UPDATE action SET status =
         'Invalid operation. No action taken.'
         WHERE CURRENT OF c1;

   END IF;

   END LOOP;
   COMMIT;
END;
```

Output Tables

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID         BAL
----------   ---------
         1         399
         2        2000
         3         599
         4        6500
         6       20099
         7        1599

SQL> SELECT * FROM action ORDER BY time_tag;

ACCOUNT_ID  O  NEW_VALUE STATUS                    TIME_TAG
----------  - ---------- ------------------------  ---------
         3  u        599 Update: Success.          18-NOV-88
         6  i      20099 Insert: Success.          18-NOV-88
         5  d            Delete: Success.          18-NOV-88
         7  u       1599 Update: ID not found.     18-NOV-88
                         Value inserted.
         1  i        399 Insert: Acct exists.      18-NOV-88
                         Updated instead.
         9  d            Delete: ID not found.     18-NOV-88
        10  x            Invalid operation.        18-NOV-88
                         No action taken.
```

Sample 5. Embedded PL/SQL

The following example shows how you can embed PL/SQL in a high-level host language such as C and demonstrates how a banking debit transaction might be done.

Input Table

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID         BAL
----------   ---------
         1        1000
         2        2000
         3        1500
         4        6500
         5         500
```

PL/SQL Block in a C Program

```
/* available online in file 'sample5' */
#include <stdio.h>
   char    buf[20];
EXEC SQL BEGIN DECLARE SECTION;
   int     acct;
   double  debit;
   double  new_bal;
   VARCHAR status[65];
   VARCHAR uid[20];
   VARCHAR pwd[20];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

main()
{
   extern double atof();

   strcpy (uid.arr,"scott");
   uid.len=strlen(uid.arr);
   strcpy (pwd.arr,"tiger");
   pwd.len=strlen(pwd.arr);
```

```
   printf("\n\n\tEmbedded PL/SQL Debit Transaction Demo\n\n");
   printf("Trying to connect...");
   EXEC SQL WHENEVER SQLERROR GOTO errprint;
   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
   printf(" connected.\n");
for (;;)       /*  Loop infinitely */
{
   printf("\n** Debit which account number? (-1 to end) ");
   gets(buf);
   acct = atoi(buf);
   if (acct == -1)  /* Need to disconnect from Oracle */
   {                /* and exit loop if account is -1 */
       EXEC SQL COMMIT RELEASE;
       exit(0);
   }

   printf("   What is the debit amount? ");
   gets(buf);
   debit = atof(buf);

   /* ---------------------------------- */
   /* ----- Begin the PL/SQL block ----- */
   /* ---------------------------------- */
   EXEC SQL EXECUTE

   DECLARE
      insufficient_funds    EXCEPTION;
      old_bal               NUMBER;
      min_bal               NUMBER := 500;
   BEGIN
      SELECT bal INTO old_bal FROM accounts
         WHERE account_id = :acct;
         -- If the account doesn't exist, the NO_DATA_FOUND
         -- exception will be automatically raised.
      :new_bal := old_bal - :debit;
      IF :new_bal >= min_bal THEN
         UPDATE accounts SET bal = :new_bal
            WHERE account_id = :acct;
         INSERT INTO journal
            VALUES (:acct, 'Debit', :debit, SYSDATE);
         :status := 'Transaction completed.';
      ELSE
         RAISE insufficient_funds;
      END IF;
      COMMIT;
```

```
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         :status := 'Account not found.';
         :new_bal := -1;
      WHEN insufficient_funds THEN
         :status := 'Insufficient funds.';
         :new_bal := old_bal;
      WHEN OTHERS THEN
         ROLLBACK;
         :status := 'Error: ' || SQLERRM(SQLCODE);
         :new_bal := -1;
   END;

   END-EXEC;
   /* ------------------------------- */
   /* ----- End the PL/SQL block ----- */
   /* ------------------------------- */

   status.arr[status.len] = '\0';  /* null-terminate */
                                   /* the string     */
   printf("\n\n   Status:  %s\n", status.arr);
   if (new_bal >= 0)
      printf("   Balance is now:  $%.2f\n", new_bal);
}  /* End of loop */

errprint:
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("\n\n>>>>> Error during execution:\n");
   printf("%s\n",sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK RELEASE;
   exit(1);
}
```

Interactive Session

```
Embedded PL/SQL Debit Transaction Demo

Trying to connect... connected.

** Debit which account number? (-1 to end) 1
   What is the debit amount? 300

   Status:  Transaction completed.
   Balance is now:  $700.00
```

```
** Debit which account number? (-1 to end) 1
   What is the debit amount? 900
   Status:  Insufficient funds.
   Balance is now:  $700.00

** Debit which account number? (-1 to end) 2
   What is the debit amount? 500

   Status:  Transaction completed.
   Balance is now:  $1500.00

** Debit which account number? (-1 to end) 2
   What is the debit amount? 100

   Status:  Transaction completed.
   Balance is now:  $1400.00

** Debit which account number? (-1 to end) 99
   What is the debit amount? 100

   Status:  Account not found.

** Debit which account number? (-1 to end) -1
```

Output Tables

```
SQL> SELECT * FROM accounts ORDER BY account_id;

ACCOUNT_ID   BAL
---------- -----
         1   700
         2  1400
         3  1500
         4  6500
         5   500

SQL> SELECT * FROM journal ORDER BY date_tag;

ACCOUNT_ID  ACTION                          AMOUNT  DATE_TAG
----------  -------------------------   ----------  ---------
         1  Debit                              300  28-NOV-88
         2  Debit                              500  28-NOV-88
         2  Debit                              100  28-NOV-88
```

Sample 6. Calling a Stored Procedure

This Pro*C program connects to Oracle, prompts the user for a department number, then calls procedure `get_employees`, which is stored in package `personnel`. The procedure declares three index-by tables as `OUT` formal parameters, then fetches a batch of employee data into the index-by tables. The matching actual parameters are host arrays.

When the procedure finishes, it automatically assigns all row values in the index-by tables to corresponding elements in the host arrays. The program calls the procedure repeatedly, displaying each batch of employee data, until no more data is found.

Input Table

```
SQL> SELECT ename, empno, sal FROM emp ORDER BY sal DESC;

ENAME             EMPNO      SAL
---------- ----------- --------
KING              7839     5000
SCOTT             7788     3000
FORD              7902     3000
JONES             7566     2975
BLAKE             7698     2850
CLARK             7782     2450
ALLEN             7499     1600
TURNER            7844     1500
MILLER            7934     1300
WARD              7521     1250
MARTIN            7654     1250
ADAMS             7876     1100
JAMES             7900      950
SMITH             7369      800
```

Stored Procedure

```
/* available online in file 'sample6' */
#include <stdio.h>
#include <string.h>

typedef char asciz;
```

```
EXEC SQL BEGIN DECLARE SECTION;
   /* Define type for null-terminated strings. */
   EXEC SQL TYPE asciz IS STRING(20);
   asciz  username[20];
   asciz  password[20];
   int    dept_no;    /* which department to query */
   char   emp_name[10][21];
   char   job[10][21];
   float  salary[10];
   int    done_flag;
   int    array_size;
   int    num_ret;    /* number of rows returned */
   int    SQLCODE;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;

int print_rows();       /* produces program output      */
int sqlerror();         /* handles unrecoverable errors */

main()
{
   int i;

   /* Connect to Oracle. */
   strcpy(username, "SCOTT");
   strcpy(password, "TIGER");

   EXEC SQL WHENEVER SQLERROR DO sqlerror();

   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf("\nConnected to Oracle as user: %s\n\n", username);

   printf("Enter department number: ");
   scanf("%d", &dept_no);
   fflush(stdin);

   /* Print column headers. */
   printf("\n\n");
   printf("%-10.10s%-10.10s%s\n", "Employee", "Job", "Salary");
   printf("%-10.10s%-10.10s%s\n", "--------", "---", "------");
```

```
   /* Set the array size. */
   array_size = 10;
   done_flag = 0;
   num_ret = 0;

   /* Array fetch loop - ends when NOT FOUND becomes true. */
   for (;;)
   {
      EXEC SQL EXECUTE
         BEGIN personnel.get_employees
            (:dept_no, :array_size, :num_ret, :done_flag,
            :emp_name, :job, :salary);
         END;
      END-EXEC;

      print_rows(num_ret);

      if (done_flag)
         break;
   }

   /* Disconnect from Oracle. */
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
}

print_rows(n)
int n;
{
   int i;

   if (n == 0)
   {
      printf("No rows retrieved.\n");
      return;
   }

   for (i = 0; i < n; i++)
      printf("%10.10s%10.10s%6.2f\n",
         emp_name[i], job[i], salary[i]);
}
```

```
sqlerror()
{
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("\nOracle error detected:");
   printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}
```

Interactive Session

```
Connected to Oracle as user: SCOTT

Enter department number: 20

Employee  Job       Salary
--------  ---       ------
SMITH     CLERK     800.00
JONES     MANAGER   2975.00
SCOTT     ANALYST   3000.00
ADAMS     CLERK     1100.00
FORD      ANALYST   3000.00
```

B

CHAR versus VARCHAR2 Semantics

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.

Major 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. So, information about trailing blanks 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 ...
```

ANSI/ISO SQL requires that two character values being compared have equal lengths. So, if both values in a comparison have datatype CHAR, *blank-padding* semantics are used. That is, before comparing character values of unequal length, PL/SQL blank-pads the shorter value to the length of the longer value. For example, given the declarations

```
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. That is, 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 one value in a comparison has datatype VARCHAR2 and the other value has datatype CHAR, 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. So, 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. So, if both values in a comparison are literals, blank-padding semantics are used. If one value is a literal, blank-padding semantics are used only if the other value has datatype 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

PL/SQL Wrapper

This appendix shows you how to run the PL/SQL Wrapper, a stand-alone utility that converts PL/SQL source code into portable object code. You can use the Wrapper to deliver PL/SQL applications without exposing your source code.

Major Topics

Advantages of Wrapping

The PL/SQL Wrapper converts PL/SQL source code into an intermediate form of object code. By hiding application internals, the Wrapper prevents

- misuse of your application by other developers
- exposure of your algorithms to business competitors

Wrapped code is as portable as source code. The PL/SQL compiler recognizes and loads wrapped compilation units automatically. Other advantages include

- platform independence—you need not deliver multiple versions of the same compilation unit
- dynamic loading—users need not shut down and relink to add a new feature
- dynamic binding—external references are resolved at load time
- strict dependency checking—invalidated program units are recompiled automatically
- normal importing and exporting—the Import/Export utility accepts wrapped files

Running the PL/SQL Wrapper

To run the PL/SQL Wrapper, enter the `wrap` command at your operating system prompt using the following syntax:

```
wrap iname=input_file [oname=output_file]
```

Leave no space around the equal signs because spaces delimit individual arguments.

The `wrap` command requires only one argument, which is

```
iname=input_file
```

where `input_file` is the path and name of the Wrapper input file. You need not specify the file extension because it defaults to `sql`. For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql
```

However, you can specify a different file extension as the following example shows:

```
wrap iname=/mydir/myfile.src
```

Optionally, the `wrap` command takes a second argument, which is

```
oname=output_file
```

where `output_file` is the path and name of the Wrapper output file. You need not specify the output file because its name defaults to that of the input file and its extension defaults to `plb` (PL/SQL binary). For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

However, you can use the option `oname` to specify a different file name and extension, as the following example shows:

```
wrap iname=/mydir/myfile oname=/yourdir/yourfile.obj
```

Input and Output Files

The input file can contain any combination of SQL statements. However, the PL/SQL Wrapper wraps only the following `CREATE` statements, which define object types, packages, or stand-alone subprograms:

```
CREATE [OR REPLACE] TYPE type_name ... OBJECT
CREATE [OR REPLACE] TYPE BODY type_name
CREATE [OR REPLACE] PACKAGE package_name
CREATE [OR REPLACE] PACKAGE BODY package_name
CREATE [OR REPLACE] FUNCTION function_name
CREATE [OR REPLACE] PROCEDURE procedure_name
```

All other SQL statements are passed intact to the output file. Comment lines are deleted unless they appear inside an object type, package, or subprogram.

When wrapped, an object type, package, or subprogram has the form

```
<header> wrapped <body>
```

where `header` begins with the reserved word `CREATE` and ends with the name of the object type, package, or subprogram, and `body` is an intermediate form of object code. The word `wrapped` tells the PL/SQL compiler that the object type, package, or subprogram was processed by the Wrapper.

The header can contain comments. For example, the Wrapper converts

```
CREATE PACKAGE
-- Author: J. Hollings
-- Date:   12/15/98
banking AS
   minimum_balance CONSTANT REAL := 25.00;
   insufficient_funds EXCEPTION;
END banking;
```

into

```
CREATE PACKAGE
-- Author: J. Hollings
-- Date:   12/15/98
banking wrapped
0
abcd ...
```

Generally, the output file is much larger than the input file.

Suggestion: When wrapping a package (or object type), wrap only the body, not the spec. That way, you give other developers all the information (subprogram specs) they need to use the package without exposing its implementation.

Error Handling

If your input file contains syntax errors, the PL/SQL Wrapper detects and reports them. However, the Wrapper cannot detect semantic errors because it does not resolve external references. For example, the Wrapper does not report the following error (*table or view does not exist*):

```
CREATE PROCEDURE raise_salary (emp_id INTEGER, amount NUMBER) AS
BEGIN
   UPDATE amp  -- should be emp
      SET sal = sal + amount WHERE empno = emp_id;
END;
```

The PL/SQL compiler resolves external references. So, semantic errors are reported when the Wrapper output file (`.plb` file) is compiled.

D

Name Resolution

This appendix explains how PL/SQL resolves references to names in potentially ambiguous SQL and procedural statements.

Major Topics

What Is Name Resolution?

During compilation, the PL/SQL compiler associates identifiers such as the name of a variable with an address (memory location), datatype, or actual value. This process is called *binding*. The association lasts through all subsequent executions until a recompilation occurs, which might cause a rebinding.

Before binding the names, PL/SQL must resolve all references to them in the compilation unit. This process is called *name resolution*. PL/SQL considers all names to be in the same namespace. So, one declaration or definition in an inner scope can hide another in an outer scope. In the following example, the declaration of variable `client` hides the definition of datatype `Client` because PL/SQL is not case sensitive except within string literals:

```
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;        -- illegal; Client resolves to the
                               -- variable client
         lead2  block1.Client; -- OK; refers to type Client
      BEGIN
         NULL;
      END;
   END;
END;
```

However, you can still refer to datatype `Client` by qualifying the reference with block label `block1`.

In the `CREATE TYPE person1` statement below, the compiler resolves the second reference to `manager` as the name of the attribute you are trying to declare. In the `CREATE TYPE person2` statement, the compiler resolves the second reference to `manager` as the name of the attribute you just declared. In both cases, the reference to `manager` generates an error because the compiler expects a type name.

```
CREATE TYPE manager AS OBJECT (dept NUMBER);
CREATE TYPE person1 AS OBJECT (manager manager);
CREATE TYPE person2 AS OBJECT (manager NUMBER, mgr manager);
```

Various Forms of References

During name resolution, the compiler can encounter various forms of references including simple unqualified names, dot-separated chains of identifiers, indexed components of a collection, and so on. Some examples of legal references follow:

```
CREATE PACKAGE pack1 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
/
CREATE PACKAGE BODY pack1 AS
   FUNCTION f1 (p1 NUMBER) RETURN t1 IS
      n NUMBER;
   BEGIN
      ...
      n := m;            -- (1) unqualified name
      n := pack1.m;      -- (2) dot-separated chain of identifiers
                         --     (package name used as scope
                         --     qualifier followed by variable name)
      n := pack1.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 := pack1.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.pack1.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
   END;
   FUNCTION f2 (q1 NUMBER) RETURN t2 IS
   BEGIN
     NULL;
   END;
END;
/
CREATE OR REPLACE PACKAGE BODY pack1 AS
   FUNCTION f1 (p1 NUMBER) RETURN t1 IS
      n NUMBER;
   BEGIN
      n := scott.pack1.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;

   FUNCTION f2 (q1 NUMBER) RETURN t2 IS
   BEGIN
      NULL;
   END;
END;
/
```

Name-Resolution Algorithm

Let us take a look at the name-resolution algorithm.

The first part of name resolution involves finding the basis. The *basis* is the smallest prefix to a dot-separated chain of identifiers that can be resolved by looking in the current scope, then moving outward to schema-level scopes.

In the previous examples, the basis for (3) `pack1.f1.p1` is `pack1`, the basis for (4) `scott.pack1.f1.n` is `scott.pack1`, and the basis for (5) `v1.a` is `v1`. In (5), the `a` in `v1.a` is a component selector and resolves as field `a` of variable `v1` because `v1` is of type `t1`, which has a field called `a`.

If a basis is not found, the compiler generates a *not declared* error. If the basis is found, the compiler attempts to resolve the complete reference. If it fails, the compiler generates an error.

The length of the basis is always 1, 2, or 3. And, it can be 3 only inside SQL scope when the compiler resolves a three-part name as

```
schema_name.table_name.column_name
```

Here are more examples of bases:

```
variable_name
type_name
package_name
schema_name.package_name
schema_name.function_name
table_name
table_name.column_name
schema_name.table_name
schema_name.table_name.column_name
```

Finding the Basis

Now, let us look at the algorithm for finding the basis.

If the compiler is resolving a name in SQL scope (which includes everything in a DML statement except items in the `INTO` clause and schema-level table names), it first attempts to find the basis in that scope. If it fails, it attempts to find the basis in PL/SQL local scopes and at the schema level just as it would for names in non-SQL scopes.

Here are the rules for finding the basis in SQL scope when the compiler expects to find a column name:

- Given one identifier, the compiler attempts to find a basis of length 1 using the identifier as an unqualified column name in one of the tables listed in any `FROM` clauses that are in scope, starting with the current scope and moving outward.
- Given a of chain two identifiers, the compiler attempts to find a basis of length 2 using the identifiers as a column name qualified by a table name or table alias, starting with the current scope and moving outward.
- Given a chain of three identifiers, the compiler attempts to find in each scope that it searches, starting with the current scope and moving outward, either
  - a basis of length 3 using the three identifiers as a column name qualified by a table name qualified by a schema name, or
  - a basis of length 2 using the first two identifiers as a column name of some user-defined type qualified by a table alias
- Given a chain of four identifiers, the compiler attempts to find a basis of length 2, using the first two identifiers as a column name of some user-defined type qualified by a table alias, starting with the current scope and moving outward.

Once the compiler finds the basis as a column name, it attempts to resolve the complete reference by finding a component of the basis and so on depending upon the type of column name.

Here are the rules for finding the basis in SQL scope when the compiler expects to find a row expression (which is a table alias that can appear by itself; it can be used only with an object table and operator `REF` or `VALUE`, or in an `INSERT` or `UPDATE` statement for an object table):

- Given one identifier, the compiler attempts to find a basis of length 1 as a table alias, starting with the current scope and moving outward. If the table alias does not correspond to an object table, the compiler generates an error.
- Given a chain of two or more identifiers, the compiler generates an error.

If the name being resolved either

- does not appear in SQL scope, or
- appears in SQL scope but the compiler cannot find a basis for it in that scope

the compiler attempts to find the basis by searching all PL/SQL scopes local to the compilation unit, starting with the current scope and moving outward. If the name is found, the length of the basis is 1. If the name is not found, the compiler attempts to find the basis by searching for schema objects using the following rules:

1. First, the compiler attempts to find a basis of length 1 by searching the current schema for a schema object whose name matches the first identifier in the chain of identifiers. The schema object found might be a package specification, function, procedure, table, view, sequence, synonym, or schema-level datatype. If it is a synonym, the basis will be resolved as the base object designated by the synonym.
2. If the previous search fails, the compiler attempts to find a basis of length 1 by searching for a public synonym whose name matches the first identifier in the chain. If this succeeds, the basis will be resolved as the base object designated by the synonym.
3. If the previous search fails and there are at least two identifiers in the chain, the compiler attempts to find a basis of length 2 by searching for a schema object whose name matches the second identifier in the chain and which is owned by a schema whose name matches the first identifier in the chain.
4. If the compiler finds a basis as a schema object, it checks the privileges on the base object. If the base object is not visible, the compiler generates a *not declared* error because an *insufficient privileges* error would acknowledge the existence of the object, which is a security violation.
5. If the compiler fails to find a basis by searching for schema objects, it generates a *not declared* error.
6. If the compiler finds a basis, it attempts to resolve the complete reference depending on how the basis was resolved. If it fails to resolve the complete reference, the compiler generates an error.

Understanding Capture

When a declaration or type definition in another scope prevents the compiler from resolving a reference correctly, that declaration or definition is said to "capture" the reference. Usually this is the result of migration or schema evolution. There are three kinds of capture: inner, same-scope, and outer. Inner and same-scope capture apply only in SQL scope.

Inner Capture

An inner capture occurs when a name in an inner scope that once resolved to an entity in an outer scope, either

- gets resolved to an entity in an inner scope, or
- causes an error because the basis of the identifier chain got captured in an inner scope and the complete reference could not be resolved

If the situation was resolved without error in an inner scope, the capture might occur unbeknown to you. Consider, the following example:

```
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 this 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`. If you add a column named `col2` to table `tab2`, as follows

```
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`.

The use of collections and object types allows for more inner capture situations. Consider the following example:

```
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 this example, the reference to `s.tab2.a` resolves to attribute `a` of column `tab2` in table `tab1` via table alias `s` which is visible in the outer scope of the query. Suppose you 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 Capture" on page D-10. According to those rules, you should revise the above 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 in the same scope, and that column has the same name as a column in the other table. Consider the following query (and refer to the previous example):

```
PROCEDURE proc IS
    CURSOR c1 IS SELECT * FROM tab1, tab2 WHERE col2 = 10;
```

In this example, the reference to `col2` in the query binds to column `col2` in table `tab1`. If you add a column named `col2` to table `tab2`, the query compiles with errors. Thus, the reference to `col2` is captured by an error.

Outer Capture

An outer capture occurs when a name in an inner scope, which once resolved to an entity in an inner scope, gets resolved to an entity in an outer scope. Fortunately, SQL and PL/SQL are designed to prevent outer captures.

Avoiding Capture

You can avoid inner capture in DML statements by following these rules:

- Specify an alias for each table in the DML statement.
- Keep table aliases unique throughout the DML statement.
- Avoid table aliases that match schema names used in the query.
- Qualify each column reference with the table alias.

Qualifying a reference with <schema-name>.<table-name> does not prevent inner capture if the DML statement references tables that have columns of a user-defined object type.

Accessing Attributes and Methods

Columns of a user-defined object type allow for more inner capture situations. To minimize problems, the name-resolution algorithm includes the following rules:

- All references to attributes and methods must be qualified by a table alias. So, when referencing a table, if you reference the attributes or methods of an object stored in that table, the table name must be accompanied by an alias. As the following examples show, column-qualified references to an attribute or method are illegal if they are prefixed with a table name (or schema and table name):

```
CREATE TYPE t1 AS OBJECT (x NUMBER);
CREATE TABLE tb1 (col t1);
SELECT col.x FROM tb1;                    -- illegal
SELECT tb1.col.x FROM tb1;                -- illegal
SELECT scott.tb1.col.x FROM scott.tb1;    -- illegal
SELECT t.col.x FROM tb1 t;
UPDATE tb1 SET col.x = 10;                -- illegal
UPDATE scott.tb1 SET scott.tb1.col.x=10;  -- illegal
UPDATE tb1 t set t.col.x = 1;
DELETE FROM tb1 WHERE tb1.col.x = 10;     -- illegal
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;                    -- illegal
SELECT REF(o) FROM ot1 o;
SELECT VALUE(ot1) FROM ot1;                  -- illegal
SELECT VALUE(o) FROM ot1 o;
DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10)); -- illegal
DELETE FROM ot1 o WHERE VALUE(o) = (t1(10));
UPDATE ot1 SET ot1 = ...                     -- illegal
UPDATE ot1 o SET o = ....
```

  The following ways to insert into an object table are legal and do not require an alias because there is no column list:

```
INSERT INTO ot1 VALUES (t1(10)); -- no row expression
INSERT INTO ot1 VALUES (10);     -- no row expression
```

Calling Subprograms and Methods

You can call a parameterless subprogram with or without an empty parameter list. Likewise, within PL/SQL scopes, the empty parameter list is optional. However, within SQL scopes, it is required.

Example 1

```
CREATE FUNCTION func1 RETURN NUMBER AS
   BEGIN
      RETURN 10;
   END;

CREATE PACKAGE pkg1 AS
    FUNCTION func1 RETURN NUMBER;
    PRAGMA RESTRICT_REFERENCES(func1,WNDS,RNDS,WNPS,RNPS);
END;
```

```
CREATE PACKAGE BODY pkg1 AS
   FUNCTION func1 RETURN NUMBER IS
   BEGIN
      RETURN 20;
   END;
END;

SELECT func1 FROM dual;
SELECT func1() FROM dual;
SELECT pkg1.func1 FROM dual;
SELECT pkg1.func1() FROM dual;

DECLARE
   x NUMBER;
BEGIN
   x := func1;
   x := func1();
   SELECT func1 INTO x FROM dual;
   SELECT func1() INTO x FROM dual;
   SELECT pkg1.func1 INTO x FROM dual;
   SELECT pkg1.func1() INTO x FROM dual;
END;
```

Example 2

```
CREATE OR REPLACE TYPE type1 AS OBJECT (
   a NUMBER,
   MEMBER FUNCTION f RETURN number,
   PRAGMA RESTRICT_REFERENCES(f,WNDS,RNDS,WNPS,RNPS)
);

CREATE TYPE BODY type1 AS
   MEMBER FUNCTION f RETURN number IS
   BEGIN
      RETURN 1;
   END;
END;

CREATE TABLE tab1 (col1 type1);
INSERT INTO tab1 VALUES (type1(10));

SELECT x.col1.f   FROM tab1 x;  -- illegal
SELECT x.col1.f() FROM tab1 x;
```

```
DECLARE
   n NUMBER;
   y type1;
BEGIN
   /* In PL/SQL scopes, an empty parameter list is optional. */
   n := y.f;
   n := y.f();
   /* In SQL scopes, an empty parameter list is required. */
   SELECT x.col1.f INTO n FROM tab1 x;    -- illegal
   SELECT x.col1.f() INTO n FROM tab1 x;
   SELECT y.f INTO n FROM tab1 x;         -- illegal
   SELECT y.f() INTO n FROM tab1 x;l
END;
```

SQL versus PL/SQL

The name-resolution rules for SQL and PL/SQL are similar. However, there are a few minor differences, which are not noticeable if you follow the capture avoidance rules.

For compatibility, the SQL rules are more permissive than the PL/SQL rules. That is, the SQL rules, which are mostly context sensitive, recognize as legal more situations and DML statements than the PL/SQL rules do.

E

Reserved Words

The words listed in this appendix are reserved by PL/SQL. That is, they have a special syntactic meaning to PL/SQL. So, you should not use them to name program objects such as constants, variables, or cursors. Some of these words (marked by an asterisk) are also reserved by SQL. So, you should not use them to name schema objects such as columns, tables, or indexes.

```
ALL*            DESC*       ISOLATION     OUT           SQLERRM
ALTER*          DISTINCT*   JAVA          PACKAGE       START*
AND*            DO          LEVEL*        PARTITION     STDDEV
ANY*            DROP*       LIKE*         PCTFREE*      SUBTYPE
ARRAY           ELSE*       LIMITED       PLS_INTEGER   SUCCESSFUL*
AS*             ELSIF       LOCK*         POSITIVE      SUM
ASC*            END         LONG*         POSITIVEN     SYNONYM*
AUTHID          EXCEPTION   LOOP          PRAGMA        SYSDATE*
AVG             EXCLUSIVE*  MAX           PRIOR*        TABLE*
BEGIN           EXECUTE     MIN           PRIVATE       THEN*
BETWEEN*        EXISTS*     MINUS*        PROCEDURE     TIME
BINARY_INTEGER  EXIT        MINUTE        PUBLIC*       TIMESTAMP
BODY            EXTENDS     MLSLABEL*     RAISE         TO*
BOOLEAN         FALSE       MOD           RANGE         TRIGGER*
BULK            FETCH       MODE*         RAW*          TRUE
BY*             FLOAT*      MONTH         REAL          TYPE
CHAR*           FOR*        NATURAL       RECORD        UID*
CHAR_BASE       FORALL      NATURALN      REF           UNION*
CHECK*          FROM*       NEW           RELEASE       UNIQUE*
CLOSE           FUNCTION    NEXTVAL       RETURN        UPDATE*
CLUSTER*        GOTO        NOCOPY        REVERSE       USE
COLLECT         GROUP*      NOT*          ROLLBACK      USER*
COMMENT*        HAVING*     NOWAIT*       ROW*          VALIDATE*
COMMIT          HEAP        NULL*         ROWID*        VALUES*
COMPRESS*       HOUR        NUMBER*       ROWLABEL      VARCHAR*
CONNECT*        IF          NUMBER_BASE   ROWNUM*       VARCHAR2*
CONSTANT        IMMEDIATE*  OCIROWID      ROWTYPE       VARIANCE
CREATE*         IN*         OF*           SAVEPOINT     VIEW*
CURRENT*        INDEX*      ON*           SECOND        WHEN
CURRVAL         INDICATOR   OPAQUE        SELECT*       WHENEVER*
CURSOR          INSERT*     OPEN          SEPERATE      WHERE*
DATE*           INTEGER*    OPERATOR      SET*          WHILE
DAY             INTERFACE   OPTION*       SHARE*        WITH*
DECLARE         INTERSECT*  OR*           SMALLINT*     WORK
DECIMAL*        INTERVAL    ORDER*        SPACE         WRITE
DEFAULT*        INTO*       ORGANIZATION  SQL           YEAR
DELETE*         IS*         OTHERS        SQLCODE       ZONE
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T