PL/SQL™ User's Guide and Reference

Release 2.3

Part No. A32542–1

ORACLE®

PL/SQL™ User's Guide and Reference, Release 2.3

Part No. A32542–1

Primary Author: Tom Portfolio

Contributors: Cailein Barclay, Gray Clossman, Ken Jacobs, Ken Jensen, Chris Jones, Cetin Ozbutun, Olga Peschansky, Dave Posner, Ken Rudin, Phil Shaw, Tim Smith, Scott Urman, Kevin Wharton

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

Preface

PL/SQL is Oracle Corporation's procedural language extension to SQL, the standard data access language for relational databases. PL/SQL offers modern software engineering features such as data encapsulation, information hiding, overloading, and exception handling, and so brings state–of–the–art programming to the Oracle Server and Toolset.

Designed to meet the practical needs of software developers, 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. Here is all the information you need to understand PL/SQL and use it effectively to solve your information management problems.

Audience

Anyone developing applications for the Oracle Server 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 the following subjects:

- a procedural programming language such as Ada, C, Cobol, Fortran, Pascal, or PL/I
- the SQL database language
- Oracle concepts and terminology
- Oracle application development tools

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.

What's New in This Edition?

Release 2.3 of PL/SQL offers an array of new features to help you build better database applications. For example, now you can benefit from

- support for file I/O
- PL/SQL table improvements such as PL/SQL table attributes and support for PL/SQL tables of records
- cursor variable improvements such as weak REF CURSOR types and support for cursor attributes
- a new fast–integer datatype
- full support for subqueries
- a new remote dependency mode

For more information, see Appendix A.

How This Guide Is Organized

The *PL/SQL User's Guide and Reference* is divided into three parts: a user's guide, a language reference, and appendices.

User's Guide

This part introduces you to PL/SQL and shows you how to use its many features.

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, expressions, assignments, block structure, declarations, scope, and built–in functions.

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: PL/SQL Tables and User–Defined Records
This chapter focuses on the composite datatypes TABLE and RECORD, which can store collections of data. You learn how to reference and manipulate these collections as whole objects.

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 the consistency of 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, which aid application development by isolating operations. It discusses procedures, functions, forward declarations, actual versus formal parameters, positional and named notation, parameter modes, parameter default values, aliasing, overloading, and recursion.

Chapter 8: Packages
This chapter shows you how to bundle related PL/SQL types, objects, 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: Execution Environments
This chapter shows you how to use PL/SQL in the SQL*Plus, Oracle Precompiler, and Oracle Call Interface (OCI) environments.

Language Reference

This part serves as a reference guide to PL/SQL commands, syntax, and semantics.

Chapter 10: Language Elements
This chapter uses BNF–style syntax definitions 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.

Appendices

This part provides a survey of new features, sample programs, supplementary technical information, and a list of reserved words.

Appendix A: New Features
This appendix looks at the array of new features offered by release 2.3 of PL/SQL.

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

Appendix C: CHAR versus VARCHAR2 Semantics
This appendix explains the subtle but important semantic differences between the CHAR and VARCHAR2 base types.

Appendix D: PL/SQL Wrapper
This appendix shows you how to run the PL/SQL Wrapper, a standalone utility that enables you to deliver PL/SQL applications without exposing your source code.

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

Notational Conventions

This guide uses the following notation in code examples:

< >	Angle brackets enclose the name of a syntactic element.
– –	A double hyphen begins a single–line comment, which extends to the end of a line.
/* and */	A slash–asterisk and an asterisk–slash delimit a multi–line comment, which can span multiple lines.
.	A dot separates an object name from a component name and so qualifies a reference.
...	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.
UPPERCASE	Uppercase denotes PL/SQL keywords.
lowercase	Lowercase denotes user–defined items such as variables, parameters, and exceptions.

The syntax of PL/SQL is described using a simple variant of Backus–Naur Form (BNF). See "Reading Syntax Definitions" on page 10 – 3.

Terms being defined for the first time, words being emphasized, error messages, and book titles are *italicized*.

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  CHAR(14),
     loc    CHAR(13))

CREATE TABLE emp
    (empno    NUMBER(4) NOT NULL,
     ename    CHAR(10),
     job      CHAR(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 appreciate your comments. As we evaluate and revise our documentation, your opinions are the most important feedback we receive. At the back of our printed manuals is a Reader's Comment Form, which we encourage you to use. If the form is not available, please use the following address or fax number:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
Fax: (415) 506–7200

Contents

PART

I User's Guide

CHAPTER

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.

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 objects can be declared. Once declared, objects can be manipulated in the executable part. Exceptions raised during execution can be dealt with in the exception–handling part.

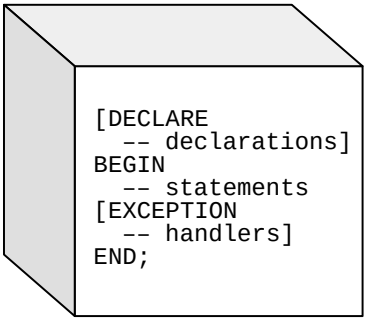

Figure 1 – 1 Block Structure

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, and NUMBER, or any PL/SQL datatype, such as BOOLEAN and 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 records and PL/SQL tables using the RECORD and TABLE 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.

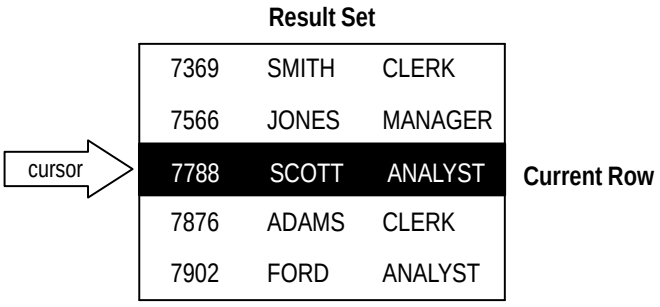

Figure 1 – 2 Query Processing

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 in a database table, 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;
END;
```

You use dot notation to reference individual fields in the record.

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 packaged procedure opens the cursor variable *generic_cv* for the chosen query:

```
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;
```

Attributes

PL/SQL variables and cursors have *attributes*, which are properties that let you reference the datatype and structure of an object without repeating its definition. Database columns and tables have similar attributes, which you can use to ease maintenance.

%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* having the same datatype as the column *title*, you 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 in 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.

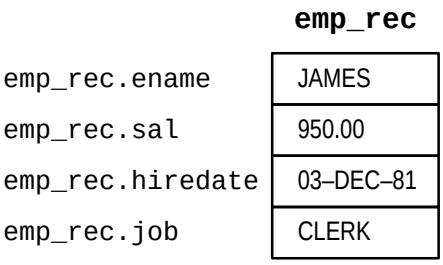

Figure 1 – 3 %ROWTYPE Record

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 yields TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition yields 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 yields 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;
```

PL/SQL Tables

Like an array, a PL/SQL table is an ordered collection of elements of the same type. Each element has a unique index number that determines its position in the ordered collection. But, unlike an array, a PL/SQL table is unbounded. So, its size can increase dynamically. Also, it does not require consecutive index numbers. So, it can be indexed by any series of integers.

PL/SQL tables help you move bulk data. They can store columns or rows of Oracle data, and they can be passed as parameters. So, PL/SQL tables make it easy to move collections of data into and out of database tables or between client–side applications and stored subprograms.

You can use a cursor FOR loop to fetch an entire column or table of Oracle data into a PL/SQL table. In the following example, you fetch a table of data into the PL/SQL table *dept_tab*:

```
DECLARE
   TYPE DeptTabTyp IS TABLE OF dept%ROWTYPE
      INDEX BY BINARY_INTEGER;
   dept_tab DeptTabTyp;
   n BINARY_INTEGER := 0;
BEGIN
   FOR dept_rec IN (SELECT * FROM dept) LOOP
      n := n + 1;
      dept_tab(n) := dept_rec;
   END LOOP;
   ...
END;
```

User–Defined 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 TimeTyp IS RECORD (minute SMALLINT, hour SMALLINT);
   TYPE MeetingTyp IS RECORD (
      day     DATE,
      time    TimeTyp,  -- nested record
      place   VARCHAR2(20),
      purpose VARCHAR2(50));
```

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

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 unit*s. Besides blocks and subprograms, PL/SQL provides the package, which allows you to group related program objects 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, program objects, 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.

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 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 types, program objects, and subprograms are public or private. Thus, packages enforce data encapsulation by letting you put type declarations in a black box. A private type 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
   salary       NUMBER(7,2);
   commission   NUMBER(7,2);
   comm_missing EXCEPTION;  -- declare exception
BEGIN
   SELECT sal, comm INTO salary, commission FROM emp
      WHERE empno = :emp_id;
   IF commission IS NULL THEN
      RAISE comm_missing;  -- raise exception
   ELSE
       :bonus := (salary * 0.05) + (commission * 0.15);
   END IF;
EXCEPTION  -- begin exception handlers
   WHEN comm_missing THEN
       -- process error
END;
```

The variables *emp_id* and *bonus* are declared and assigned values in a host environment. For more information about host variables, see "Oracle Precompiler Environment" on page 9 – 7.

Architecture

The PL/SQL runtime system is a technology, not an independent product. Think of this technology as an engine that 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 might be available in the Oracle Server but unavailable in tools, or the other way around. 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.

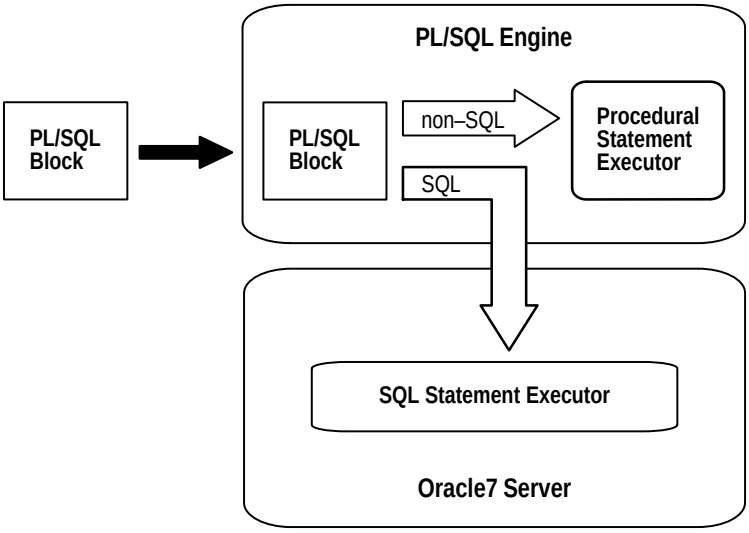

Figure 1 – 4 PL/SQL Engine

The PL/SQL engine executes procedural statements but sends SQL statements to the SQL Statement Executor in the Oracle Server.

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 Server Manager, lacking a local PL/SQL engine, must send anonymous blocks to Oracle.

Stored Subprograms

Named PL/SQL blocks (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 database object, which can be referenced by any number of applications connected to that database.

Stored subprograms defined within a package are called *packaged* subprograms; those defined independently are called *standalone* subprograms. (Subprograms defined within another subprogram or within a PL/SQL block are called *local* subprograms. They 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 Server Manager. For example, you might call the standalone procedure *create_dept* from SQL*Plus as follows:

```
SQL> EXECUTE 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. Moreover, stored subprograms take advantage of the Oracle shared memory capability. 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. 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
- higher productivity
- better performance
- portability
- integration with Oracle

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 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.

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.

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.

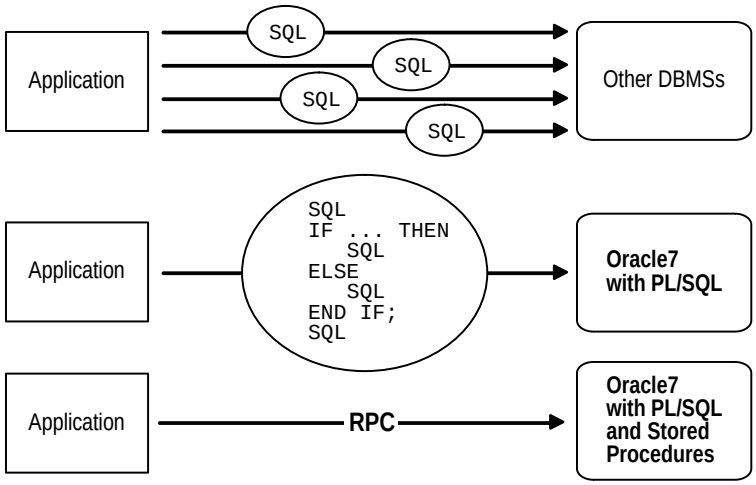

Figure 1 – 5 PL/SQL Boosts Performance

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 Oracle. This saves time and reduces network traffic.

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.

Integration with Oracle

Both PL/SQL and Oracle are based on SQL. Moreover, PL/SQL supports all the SQL datatypes. Combined with the direct access that SQL provides, these shared datatypes integrate PL/SQL with the Oracle data dictionary.

The %TYPE and %ROWTYPE attributes further integrate PL/SQL with the data dictionary. 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 at run time. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.

CHAPTER

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.

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

PL/SQL is not case sensitive, so lowercase letters are equivalent to corresponding uppercase 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. For example, 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 the following 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

Simple symbols consist of one character; a list follows:

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

Compound Symbols

Compound symbols consist of two characters; a list follows:

**	exponentiation operator
<>	relational operator
!=	relational operator
~=	relational operator
^=	relational operator
<=	relational operator
>=	relational operator
:=	assignment operator
=>	association operator
..	range operator
||	concatenation operator
<<	(beginning) label delimiter
>>	(ending) label delimiter
––	single–line comment indicator
/*	(beginning) multi–line comment delimiter
*/	(ending) multi–line comment delimiter

Identifiers

You use identifiers to name PL/SQL program objects 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 legal:

```
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, use meaningful names such as *credit_limit* and *cost_per_thousand.* Avoid obscure names such as *cr_lim* and *cpm*.

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 legal:

```
"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.

Using PL/SQL reserved words as quoted identifiers is allowed but *not* recommended. It is poor programming practice to reuse reserved words.

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:

$$5E3 = 5 \times 10^3 = 5 \times 1000 = 5000$$

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

$$5E-3 = 5 \times 10^{-3} = 5 \times 0.001 = 0.005$$

Character Literals

A character literal is an individual character enclosed by single quotes (apostrophes). Several examples follow:

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

Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols. PL/SQL is case sensitive within character literals. For example, PL/SQL considers the literals 'Z' and 'z' to be different.

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 and FALSE and the non–value 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. An example follows:

```
/* 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 next three examples illustrate some popular formats:

```
/* The following line computes the area of a circle using pi,
   which is the ratio between the circumference and diameter. */
area := pi * radius**2;

/*************************************************************
 * The following line computes the area of a circle using pi, *
 * which is the ratio between the circumference and diameter. *
 *************************************************************/
area := pi * radius**2;

/*
  The following line computes the area of a circle using pi,
  which is the ratio between the circumference and diameter.
*/
area := pi * radius**2;
```

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

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

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 scalar and composite datatypes. A *scalar* type has no internal components. A *composite* type has internal components that can be manipulated individually. A *reference* type contains values, called pointers, that designate other program objects.

Figure 2 – 1 shows the predefined datatypes available for your use. An additional scalar type, MLSLABEL, is available with Trusted Oracle, a specially secured version of Oracle. The scalar types fall into four families, which store number, character, date/time, or Boolean data, respectively.

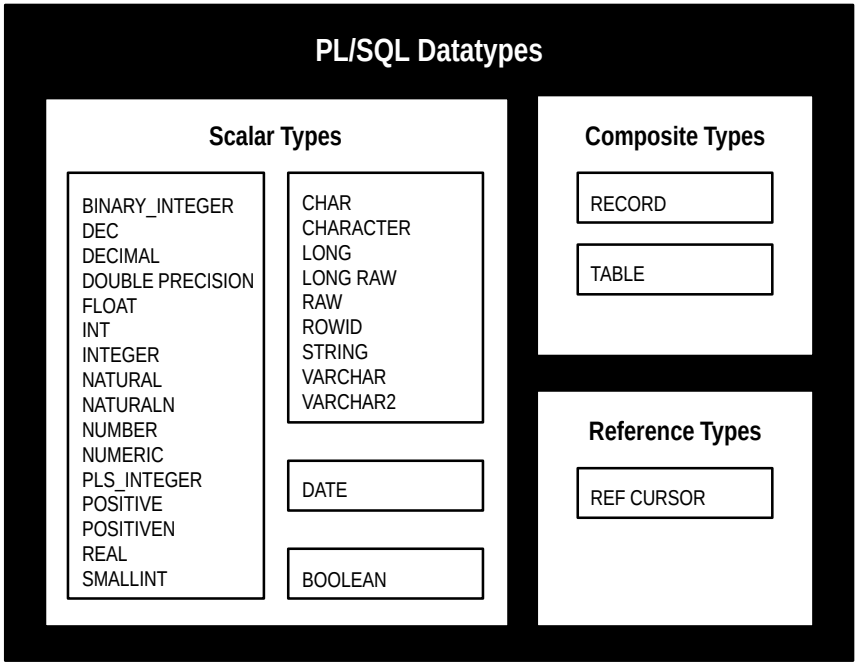

Figure 2 – 1 Predefined Datatypes

This section discusses the scalar types; the composite types are discussed in Chapter 4; the reference type is discussed in "Using Cursor Variables" on page 5 – 17.

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. For more information, see "PLS_INTEGER" on page 2 – 12.

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 (0 .. 2147483647)
- NATURALN (0 .. 2147483647)
- POSITIVE (1 .. 2147483647)
- POSITIVEN (1 .. 2147483647)

You can use these subtypes when you want to restrict a variable to non–negative integer values. The subtypes NATURALN and POSITIVEN are predefined as NOT NULL. For more information about the NOT NULL constraint, see "Using NOT NULL" on page 2 – 23.

NUMBER

You use the NUMBER datatype to store fixed or floating–point numbers of virtually any size. You can specify *precision*, which is the total number of digits, and *scale*, which determines where rounding occurs. The syntax follows:

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

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; the magnitude range is 1.0E–129 .. 9.99E125. If you do not specify the precision, it defaults to the maximum value supported by your system.

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

Subtypes

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

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

FLOAT is another subtype of NUMBER. However, you cannot specify a scale for FLOAT variables; you can only specify a binary precision. The maximum precision of a FLOAT value is 126 binary digits, which is roughly equivalent to 38 decimal digits.

You can use these subtypes for compatibility with ANSI/ISO and IBM types or when you want an identifier more descriptive than NUMBER.

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 are both integer types with 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.

CHAR

You use the CHAR datatype to store fixed–length (blank–padded if necessary) character data. How the data is represented internally depends on the database character set, which might be 7–bit ASCII or EBCDIC Code Page 500, for example.

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. If you do not specify the maximum length, it defaults to 1.

Remember, you specify the maximum length of a CHAR(*n*) variable in bytes, not characters. So, if a CHAR(*n*) variable stores multi–byte characters, its maximum length is less than *n* characters.

Although the maximum length of a CHAR(*n*) variable is 32767 bytes, the maximum width of a CHAR database column is 255 bytes. Therefore, you cannot insert values longer than 255 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 select a value longer than 32767 bytes from a LONG column into a CHAR(*n*) variable.

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

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 can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2147483647 bytes. However, you cannot select a value longer than 32760 bytes from a LONG column into a LONG 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 *Oracle7 Server 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 character data, except that PL/SQL does not interpret raw data. Likewise, Oracle does no character set conversions (from 7–bit ASCII to EBCDIC Code Page 500, for example) 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.

Although the maximum length of a RAW variable is 32767 bytes, the maximum width of a RAW database column is 255 bytes. Therefore, you cannot insert values longer than 255 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 select a value longer than 32767 bytes from a LONG RAW column into a RAW variable.

LONG RAW

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 RAW value into a LONG RAW database column because the maximum width of a LONG RAW column is 2147483647 bytes. However, you cannot select a value longer than 32760 bytes from a LONG RAW column into a LONG RAW variable.

ROWID

Internally, every Oracle database table has a ROWID pseudocolumn, which stores binary values called *rowids*. Rowids uniquely identify rows and provide the fastest way to access particular rows. You use the ROWID datatype to store rowids in a readable format. The maximum length of a ROWID variable is 256 bytes.

When you select or fetch a rowid into a ROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to an 18–byte character string and returns it in the format

```
BBBBBBBB.RRRR.FFFF
```

where BBBBBBBB is the block in the database file, RRRR is the row in the block (the first row is 0), and FFFF is the database file.

These numbers are hexadecimal. For example, the rowid

```
0000000E.000A.0007
```

points to the 11th row in the 15th block in the 7th database file.

Typically, ROWID variables are compared 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 – 47.

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.

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.

Although the maximum length of a VARCHAR2(*n*) variable is 32767 bytes, the maximum width of a VARCHAR2 database column is 2000 bytes. Therefore, you cannot insert values longer than 2000 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 select a value longer than 32767 bytes from a LONG column into a VARCHAR2(*n*) variable.

Some important semantic differences between the CHAR and VARCHAR2 base types are described in Appendix C.

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. However, the VARCHAR datatype might change to accommodate emerging SQL standards. So, it is a good idea to use VARCHAR2 rather than VARCHAR.

BOOLEAN

You use the BOOLEAN datatype to store the values TRUE and FALSE and the non–value NULL. Recall that NULL stands for a missing, unknown, or inapplicable value.

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

DATE

You use the DATE datatype to store fixed–length date values. The DATE datatype takes no parameters. Valid dates for DATE variables include January 1, 4712 BC to December 31, 4712 AD.

When stored in a database column, 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.

MLSLABEL

With Trusted Oracle, you use the MLSLABEL datatype to store variable–length, binary operating system labels. Trusted Oracle uses labels to control access to data. For more information, see *Trusted Oracle7 Server Administrator's Guide*.

You can use the MLSLABEL datatype to define a database column. Also, you can use the %TYPE and %ROWTYPE attributes to reference the column. However, with standard Oracle, such columns can store only nulls.

With Trusted Oracle, you can insert any valid operating system label into a column of type MLSLABEL. If the label is in text format, Trusted Oracle converts it to a binary value automatically. The text string can be up to 255 bytes long. However, the internal length of an MLSLABEL value is between 2 and 5 bytes.

With Trusted Oracle, you can also select values from a MLSLABEL column into a character variable. Trusted Oracle converts the internal binary value to a VARCHAR2 value automatically.

User–Defined Subtypes

Each PL/SQL base type specifies a set of values and a set of operations applicable to objects 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.

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.

Subtypes can increase reliability, provide compatibility with ANSI/ISO and IBM types, and improve readability by indicating the intended use of constants and variables.

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;
```

where *subtype_name* is a type specifier used in subsequent declarations and *base_type* stands for the following syntax:

```
{  cursor_name%ROWTYPE
 | cursor_variable_name%ROWTYPE
 | plsql_table_name%TYPE
 | record_name%TYPE
 | scalar_type_name
 | table_name%ROWTYPE
 | table_name.column_name%TYPE
 | variable_name%TYPE}
```

For example, all of the following subtype definitions are legal:

```
DECLARE
   SUBTYPE EmpDate IS DATE;           -- based on DATE type
   SUBTYPE Counter IS NATURAL;        -- based on NATURAL subtype
   TYPE NameTab IS TABLE OF VARCHAR2(10)
      INDEX BY BINARY_INTEGER;
   SUBTYPE EnameTab IS NameTab;       -- based on TABLE type
   TYPE TimeTyp IS RECORD (minute INTEGER, hour INTEGER);
   SUBTYPE Clock IS TimeTyp;          -- based on RECORD type
   SUBTYPE ID_Num IS emp.empno%TYPE;  -- based on column type
   CURSOR c1 IS SELECT * FROM dept;
   SUBTYPE Dept_Rec IS c1%ROWTYPE;    -- based on cursor rowtype
```

However, you cannot specify a 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
   SUBTYPE Word IS VARCHAR2(15);       -- illegal
```

Although you cannot define constrained subtypes directly, you can use a simple workaround to define size–constrained subtypes indirectly. Simply 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 adopts the size constraint (if any) of the column. However, the subtype does *not* adopt other kinds of constraints such as NOT NULL.

Using Subtypes

Once you define a subtype, you can declare objects of that type. In the example below, you declare two variables of type *Counter*. Notice how the subtype name indicates the intended use of the variables.

```
DECLARE
   SUBTYPE Counter IS NATURAL;
   rows      Counter;
   employees 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;
   ...
END;
```

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

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

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

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

Datatype Conversion

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

Explicit Conversion

To specify conversions explicitly, you use built–in functions that convert values from one datatype to another. Table 2 – 1 shows which function to use in a given situation. For example, to convert a CHAR value to a NUMBER value, you use the function TO_NUMBER. For more information about these functions, see *Oracle7 Server SQL Reference*.

	To				
From	**CHAR**	**DATE**	**NUMBER**	**RAW**	**ROWID**
CHAR		TO_DATE	TO_NUMBER	HEXTORAW	CHARTOROWID
DATE	TO_CHAR				
NUMBER	TO_CHAR	TO_DATE			
RAW	RAWTOHEX				
ROWID	ROWIDTOCHAR				

Table 2 – 1 Conversion Functions

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 – 2 shows which implicit conversions PL/SQL can do.

	To								
From	**BIN_INT**	**CHAR**	**DATE**	**LONG**	**NUMBER**	**PLS_INT**	**RAW**	**ROWID**	**VARCHAR2**
BIN_INT		✓		✓	✓	✓			✓
CHAR	✓		✓	✓	✓	✓	✓	✓	✓
DATE		✓		✓					✓
LONG		✓					✓		✓
NUMBER	✓	✓		✓		✓			✓
PLS_INT	✓	✓		✓	✓				✓
RAW		✓		✓					✓
ROWID		✓							✓
VARCHAR2	✓	✓	✓	✓	✓	✓	✓	✓	

Table 2 – 2 Implicit Conversions

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 PL/SQL 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, it is poor programming practice to rely on implicit datatype conversions 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.

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. For example, the following declarations are equivalent:

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

In constant declarations, 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

If you prefer, you can use the reserved word DEFAULT instead of the assignment operator to initialize variables and constants. For example, the declarations

```
tax_year SMALLINT := 95;
valid    BOOLEAN := FALSE;
```

can be rewritten as follows:

```
tax_year SMALLINT DEFAULT 95;
valid    BOOLEAN DEFAULT FALSE;
```

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.

Note, however, that a NOT NULL column constraint does *not* apply to variables declared using %TYPE. 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 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 the *c1* cursor.

```
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. In the following example, you select column values into a record named *emp_rec*:

```
DECLARE
   emp_rec emp%ROWTYPE;
   ...
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;
```

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;
```

For more information about database column aliases, see *Oracle7 Server SQL Reference.*

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 – 8.

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 objects 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, you reference the database link *newyork* because the (standalone) 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 database objects such as tables, sequences, views, standalone subprograms, and packages. However, you cannot create synonyms for objects declared within subprograms or packages. That includes constants, variables, cursors, cursor variables, exceptions, and packaged procedures.

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 – 30.

Case Sensitivity

Like other 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
   ZIP_CODE INTEGER;  -- same as zip_code and Zip_Code
```

Name Resolution

In potentially ambiguous SQL statements, the names of local variables and formal parameters take precedence over the names of database tables. For example, the following UPDATE statement fails because PL / SQL assumes that *emp* refers to the loop counter:

```
FOR emp IN 1..5 LOOP
   ...
   UPDATE emp SET bonus = 500 WHERE ...
END LOOP;
```

Likewise, the following SELECT statement fails because PL / SQL assumes that *emp* refers to the formal parameter:

```
PROCEDURE calc_bonus (emp NUMBER, bonus OUT REAL) IS
   avg_sal REAL;
BEGIN
   SELECT AVG(sal) INTO avg_sal FROM emp WHERE ...
```

In such cases, you can prefix the table name with a username, as follows, but it is better programming practice to rename the variable or formal parameter.:

```
PROCEDURE calc_bonus (emp NUMBER, bonus OUT REAL) IS
   avg_sal REAL;
BEGIN
   SELECT AVG(sal) INTO avg_sal FROM scott.emp WHERE ...
```

Unlike the names of tables, the names of 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;
                -- refers to formal parameter and local variable
```

For a full discussion of name resolution, see *Oracle7 Server Application Developer's Guide.*

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.

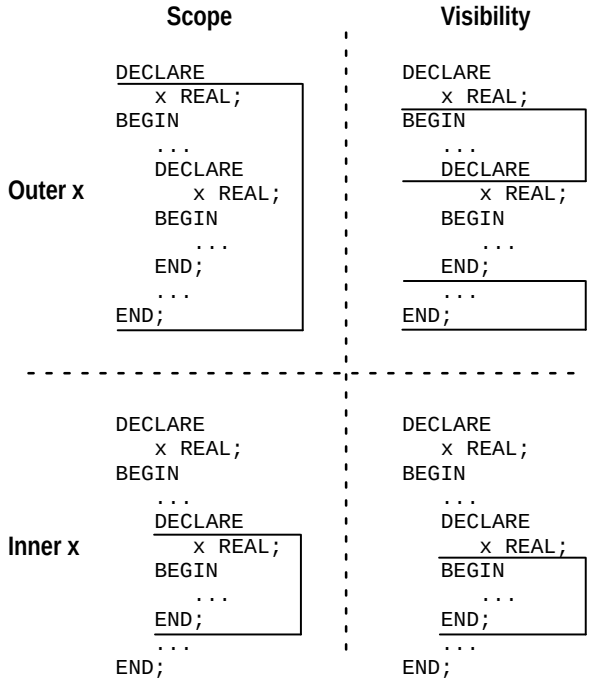

Figure 2 – 2 Scope and Visibility

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 objects 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 nested at the same level because those identifiers are neither local nor global to the block. The following example illustrates the scope rules:

```
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
```

Therefore, 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 and FALSE and the non–value 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 ...
```

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

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

Database Values

Alternatively, you can use the SELECT (or FETCH) 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:

```
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 – 3 shows the default order of operations from first to last (top to bottom).

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

Table 2 – 3 Order of Operations

Operators with higher precedence are applied first. For example, both of the following expressions yield 8 because division has a higher precedence than addition:

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

Operators with the same precedence are applied in no particular order.

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 of the truth tables in Figure 2 – 3. AND and OR are binary operators; NOT is a unary operator.

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Figure 2 – 3 Truth Tables

As the truth tables show, AND returns the value TRUE only if both its operands are true. On the other hand, OR returns the value 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 – 39.

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
```

Usually, PL/SQL stops evaluating a logical 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.

Comparison Operators

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

Relational Operators

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

Operator	*Meaning*
=	is equal to
<>, !=, ~=	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is 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 yields 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 yields 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

The concatenation operator (| |) appends one string to another. For example, the expression

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

returns the 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

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 yields TRUE:

```
number1 > number2
```

Character

Likewise, you can 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 yields 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, refer to Appendix C.

Date

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 yields 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 TRUE or FALSE, 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* has the datatype BOOLEAN, the IF statement

```
IF done = TRUE THEN ...
```

can be simplified as follows:

```
IF done THEN ...
```

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. So, if either or both *x* and *y* are null, the first IF statement assigns the value of *y* to *high*, but the second IF statement assigns the value of *x* to *high*. If neither *x* nor *y* is null, both IF statements assign the same value 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 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 more than 75 powerful functions to help you manipulate data. These built–in functions fall into the following categories:

- error–reporting
- number
- character
- conversion
- date
- miscellaneous

Table 2 – 4 shows the functions in each category.

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.

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

For descriptions of the error–reporting functions, see Chapter 10. For descriptions of the other functions, see *Oracle7 Server SQL Reference*.

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

Table 2 – 4 Built–in Functions

CHAPTER

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. And, their proper use leads naturally to a well–structured program.

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.

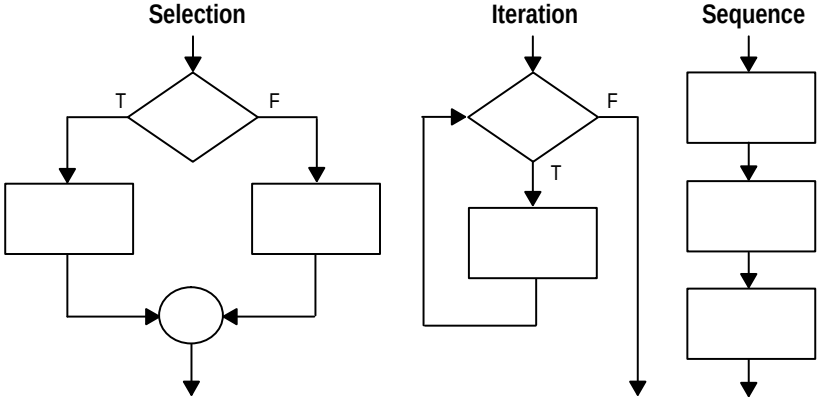

Figure 3 – 1 Control Structures

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, FALSE, or NULL). 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 yields TRUE. If the condition yields 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 yields FALSE or NULL. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first or second UPDATE statement is executed when the condition is true or false, respectively:

```
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 yields 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 yields TRUE, its associated sequence of statements is executed and control passes to the next statement. If all conditions yield 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 more 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 yields 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 the 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 – 7.

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 yields 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 yields TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition yields FALSE. When the FETCH statement fails to return a row, the condition yields 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 yields TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition yields 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. Since the condition is tested at the top of the loop, the sequence might execute zero times. In the last example, if the initial value of *total* is greater than 25000, the condition yields 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. 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, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound, as the example below shows. 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. So, the loop counter can appear in expressions 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. For example, the following iteration schemes are legal:

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

As you can see, the lower bound need not be 1. However, the loop counter increment (or decrement) must be 1. 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 following 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 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 make the meaning and action of conditional statements clear and so improve readability.

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, simply add the NULL statement, as follows:

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

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 meets this requirement. So, you can use the NULL statement 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 receive 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;
```

CHAPTER

4

PL/SQL Tables and User–Defined Records

Knowledge is that area of ignorance that we arrange and classify.

Ambrose Bierce

In Chapter 2, you learned about the PL/SQL scalar datatypes, which can store only one item of data. In this chapter, you learn about the composite datatypes TABLE and RECORD, which can store collections of data. You also learn how to reference and manipulate these collections as whole objects.

PL/SQL Tables

Objects of type TABLE are called *PL/SQL tables*, which are modeled as (but not the same as) database tables. For example, a PL/SQL table of employee names is modeled as a database table with two columns, which store a primary key and character data, respectively. Although you cannot use SQL statements to manipulate a PL/SQL table, its primary key gives you array–like access to rows. Think of the key and rows as the index and elements of a one–dimensional array.

Like an array, a PL/SQL table is an ordered collection of elements of the same type. Each element has a unique index number that determines its position in the ordered collection. However, PL/SQL tables differ from arrays in two important ways. First, arrays have fixed lower and upper bounds, but PL/SQL tables are unbounded. So, the size of a PL/SQL table can increase dynamically. Second, arrays require consecutive index numbers, but PL/SQL tables do not. This characteristic, called *sparsity*, allows the use of meaningful index numbers. For example, you can use a series of employee numbers (such as 7369, 7499, 7521, 7566, ...) to index a PL/SQL table of employee names.

Why Use PL/SQL Tables?

PL/SQL tables help you move bulk data. They can store columns or rows of Oracle data, and they can be passed as parameters. So, PL/SQL tables make it easy to move collections of data into and out of database tables or between client–side applications and stored subprograms. You can even use PL/SQL tables of records to simulate local database tables.

Also, with the Oracle Call Interface (OCI) or the Oracle Precompilers, you can bind host arrays to PL/SQL tables declared as the formal parameters of a subprogram. That allows you to pass host arrays to stored functions and procedures.

Defining TABLE Types

To create PL/SQL tables, you take two steps. First, you define a TABLE type, then declare PL/SQL tables of that type. You can define TABLE types in the declarative part of any block, subprogram, or package using the syntax

```
TYPE table_type_name IS TABLE OF datatype [NOT NULL]
   INDEX BY BINARY_INTEGER;
```

where *table_type_name* is a type specifier used in subsequent declarations of PL/SQL tables.

The INDEX BY clause must specify datatype BINARY_INTEGER, which has a magnitude range of –2147483647 .. 2147483647. If the element type is a record type, every field in the record must have a scalar datatype such as CHAR, DATE, or NUMBER.

To specify the element type, you can use %TYPE to provide the datatype of a variable or database column. In the following example, you define a TABLE type based on the *ename* column:

```
DECLARE
   TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
      INDEX BY BINARY_INTEGER;
```

The next example shows that you can add the NOT NULL constraint to a TABLE type definition and so prevent the storing of nulls in PL/SQL tables of that type:

```
DECLARE
   TYPE SalTabTyp IS TABLE OF emp.sal%TYPE NOT NULL
      INDEX BY BINARY_INTEGER;
```

An initialization clause is not required (or allowed).

You can also use %ROWTYPE to specify the element type. In the following example, you define a TABLE type based on the *emp* table:

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
```

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

```
DECLARE
   TYPE TimeRecTyp IS RECORD (
      hour   SMALLINT := 0,
      minute SMALLINT := 0,
      second SMALLINT := 0);
   TYPE TimeTabTyp IS TABLE OF TimeRecTyp
      INDEX BY BINARY_INTEGER;
```

Function Results

The example below shows that you can specify a TABLE type in the RETURN clause of a function specification. That allows the function to return a PL/SQL table of the same type.

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   ...
   FUNCTION top_n_sals (n INTEGER) RETURN EmpTabTyp IS ...
```

Declaring PL/SQL Tables

Once you define a TABLE type, you can declare PL/SQL tables of that type, as the following examples show:

```
DECLARE
   TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
      INDEX BY BINARY_INTEGER;
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   sal_tab SalTabTyp;  -- declare PL/SQL table
   emp_tab EmpTabTyp;  -- declare another PL/SQL table
```

The identifiers *sal_tab* and *emp_tab* represent entire PL/SQL tables. Each element of *sal_tab* will store an employee salary. Each element of *emp_tab* will store a whole employee record.

A PL/SQL table is unbounded; its index can include any BINARY_INTEGER value. So, you cannot initialize a PL/SQL table in its declaration. For example, the following declaration is illegal:

```
sal_tab SalTabTyp := (1500, 2750, 2000, 950, 1800);  -- illegal
```

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

As Parameters

You can declare PL/SQL tables as the formal parameters of functions and procedures. That way, you can pass PL/SQL tables to stored subprograms and from one subprogram to another. In the following example, you declare PL/SQL tables as the formal parameters of two packaged procedures:

```
PACKAGE emp_actions IS
   TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
      INDEX BY BINARY_INTEGER;
   TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
      INDEX BY BINARY_INTEGER;
   ...
   PROCEDURE hire_batch (ename_tab IN EnameTabTyp,
                         sal_tab   IN SalTabTyp, ...);
   PROCEDURE log_names (ename_tab IN EnameTabTyp);
END emp_actions;
```

To define the behavior of formal parameters, you use parameter modes. The OUT and IN OUT modes let you return values to the caller of a subprogram when you exit. 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.

Referencing PL/SQL Tables

To reference elements in a PL/SQL table, you specify an index number using the syntax

```
plsql_table_name(index)
```

where *index* is an expression that yields a BINARY_INTEGER value or a value implicitly convertible to that datatype. In the following example, you reference an element in the PL/SQL table *hiredate_tab*:

```
hiredate_tab(i + j – 1) ...
```

As the example below shows, the index number can be negative. (For an exception, see "Using Host Arrays with PL/SQL Tables" on page 4 – 15.)

```
hiredate_tab(–5) ...
```

The following example shows that you can reference the elements of a PL/SQL table in subprogram calls:

```
raise_salary(empno_tab(i), amount);  –– call subprogram
```

Assignments

You can assign one PL/SQL table to another only if they have the same datatype. For example, the following assignment is legal:

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   TYPE TempTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab1 EmpTabTyp;
   emp_tab2 EmpTabTyp;
BEGIN
   ...
   emp_tab2 := emp_tab1;  –– assign one PL/SQL table to another
```

You can assign the value of an expression to a specific element in a PL/SQL table using the following syntax:

```
plsql_table_name(index) := expression;
```

In the next example, you assign the sum of variables *salary* and *increase* to an element in the PL/SQL table *sal_tab*:

```
sal_tab(i) := salary + increase;
```

Note: Until an element is assigned a value, it does not exist. If you reference a nonexistent element, PL/SQL raises the predefined exception NO_DATA_FOUND.

PL/SQL Tables of Records

With a PL/SQL table of records, you use the following syntax to reference fields in a record:

```
plsql_table_name(index).field_name
```

For example, the following IF statement references a field in the record stored by the first element of the PL/SQL table *emp_tab*:

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab EmpTabTyp;
BEGIN
   ...
   IF emp_tab(1).job = 'CLERK' THEN ...
END;
```

Function Results

When calling a function that returns a PL/SQL table, you use the following syntax to reference elements in the table:

```
function_name(parameters)(index)
```

For example, the following call to the function *new_sals* references the third element in the PL/SQL table *sal_tab*:

```
DECLARE
   TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
      INDEX BY BINARY_INTEGER;
   salary REAL;
   FUNCTION new_sals (max_sal REAL) RETURN SalTabTyp IS
      sal_tab SalTabTyp;
   BEGIN
      ...
      RETURN sal_tab;  -- return PL/SQL table
   END;
BEGIN
   salary := new_sals(5000)(3);  -- call function
   ...
END;
```

If the function result is a PL/SQL table of records, you use the following syntax to reference fields in a record:

```
function_name(parameters)(index).field_name
```

For example, the following call to the function *new_depts* references the field *loc* in the record stored by the third element of the PL/SQL table *dept_tab*:

```
DECLARE
   TYPE DeptTabTyp IS TABLE OF dept%ROWTYPE
      INDEX BY BINARY_INTEGER;
   FUNCTION new_depts (max_num INTEGER) RETURN DeptTabTyp IS
      dept_tab DeptTabTyp;
   BEGIN
      ...
      RETURN dept_tab;
   END;
BEGIN
   ...
   IF new_depts(90)(3).loc = 'BOSTON' THEN ...
END;
```

Restriction

Currently, you cannot use the syntax above to call a parameterless function because PL/SQL does not allow empty parameter lists. That is, the following syntax is illegal:

```
function_name()(index)  -- illegal; empty parameter list
```

Instead, declare a local PL/SQL table to which you can assign the function result, then reference the PL/SQL table directly, as shown in the following example:

```
DECLARE
   TYPE JobTabTyp IS TABLE OF emp.job%TYPE
      INDEX BY BINARY_INTEGER;
   job_tab JobTabTyp;  -- declare local PL/SQL table
   job_title emp.job%TYPE;
   FUNCTION new_jobs RETURN JobTabTyp IS
      new_job_tab JobTabTyp;
   BEGIN
      ...
      RETURN new_job_tab;  -- return PL/SQL table
   END;
BEGIN
   ...
   job_tab := new_jobs;  -- assign function result
   job_title := job_tab(1);  -- reference PL/SQL table
   ...
END;
```

Using PL/SQL Table Attributes

Attributes are characteristics of an object. For example, a cursor has the attributes %FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT. Likewise, a PL/SQL table has the attributes EXISTS, COUNT, FIRST, LAST, PRIOR, NEXT, and DELETE. They make PL/SQL tables easier to use and your applications easier to maintain. To apply the attributes to a PL/SQL table, you use dot notation, as follows:

```
plsql_table_name.attribute_name
```

The attributes EXISTS, PRIOR, NEXT, and DELETE take parameters. Each parameter must be an expression that yields a BINARY_INTEGER value or a value implicitly convertible to that datatype.

DELETE acts like a procedure, which is called as a statement. However, the other PL/SQL table attributes act like a function, which is called as part of an expression.

Using EXISTS

EXISTS(*n*) returns TRUE if the *n*th element in a PL/SQL table exists. Otherwise, EXISTS(*n*) returns FALSE. You can use EXISTS to avoid the exception NO_DATA_FOUND, which is raised when you reference a nonexistent element. In the following example, PL/SQL executes the assignment statement only if the element *sal_tab(i)* exists:

```
IF sal_tab.EXISTS(i) THEN
   sal_tab(i) := sal_tab(i) + 500;
ELSE
   RAISE salary_missing;
END IF;
```

Using COUNT

COUNT returns the number of elements that a PL/SQL table contains. For example, if the PL/SQL table *ename_tab* contains 50 elements, the following IF condition is true:

```
IF ename_tab.COUNT = 50 THEN
   ...
END;
```

COUNT is useful because the future size of a PL/SQL table is unconstrained and therefore unknown. Suppose you fetch a column of Oracle data into a PL/SQL table. How many elements does the PL/SQL table contain? COUNT gives you the answer.

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

```
FOR i IN 1 .. job_tab.COUNT LOOP
   ...
END LOOP;
```

Using FIRST and LAST

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

```
IF sal_tab.FIRST = sal_tab.LAST THEN  -- sal_tab has one element
   ...
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 emp_tab.FIRST .. emp_tab.LAST LOOP
   ...
END LOOP;
```

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

```
i BINARY_INTEGER := sal_tab.FIRST;
WHILE i IS NOT NULL LOOP
   ...
   IF sal_tab(i) > 5000 THEN
      RAISE over_limit;
   END IF;
END LOOP;
```

Using PRIOR and NEXT

PRIOR(*n*) returns the index number that precedes index *n* in a PL/SQL table. NEXT(*n*) returns the index number that succeeds index *n*. If *n* has no predecessor, PRIOR(*n*) returns a null. Likewise, if *n* has no successor, NEXT(*n*) returns a null.

PRIOR and NEXT do not wrap from one end of a PL/SQL table to the other. For example, the following statement assigns a null to *n* because the first element in a PL/SQL table has no predecessor:

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

Note that PRIOR is the inverse of NEXT. For example, the following statement assigns index *n* to itself:

```
n := sal_tab.PRIOR(sal_tab.NEXT(n));  -- assigns n to n
```

You can use PRIOR or NEXT to traverse PL/SQL tables indexed by any series of integers. (Recall that index numbers need not be consecutive.) In the following example, the PL/SQL table is indexed by a series of employee numbers, which begins with 1000:

```
i BINARY_INTEGER := 1000;
WHILE i IS NOT NULL LOOP
   raise_salary(empno_tab(i));  -- pass element to procedure
   i := empno_tab.NEXT(i);  -- get index of next element
END LOOP;
```

Likewise, you can use PRIOR or NEXT to traverse PL/SQL tables from which some elements have been deleted, as the following generic example shows:

```
DECLARE
   ...
   i BINARY_INTEGER;
BEGIN
   ..
   i := any_tab.FIRST;  -- get index of first element
   WHILE i IS NOT NULL LOOP
      ...  -- process any_tab(i)
      i := any_tab.NEXT(i);  -- get index of next element
   END LOOP;
END;
```

Using DELETE

This attribute has three forms. DELETE removes all elements from a PL/SQL table. DELETE(*n*) removes the *n*th element. If *n* is null, DELETE(*n*) does nothing. DELETE(*m*, *n*) removes all elements in the range *m* .. *n*. If *m* is larger than *n* or if *m* or *n* is null, DELETE(*m*, *n*) does nothing.

DELETE lets you free the resources held by a PL/SQL table. DELETE(*n*) and DELETE(*m*, *n*) let you prune a PL/SQL table. Consider the following examples:

```
ename_tab.DELETE(3);       -- delete element 3
ename_tab.DELETE(5, 5);    -- delete element 5
ename_tab.DELETE(20, 30);  -- delete elements 20 through 30
ename_tab.DELETE(-15, 0);  -- delete elements -15 through 0
ename_tab.DELETE;          -- delete entire PL/SQL table
```

If an element to be deleted does not exist, DELETE simply skips it; no exception is raised.

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

Restriction

Currently, you cannot use PL/SQL table attributes in a SQL statement. If you try, you get a compilation error, as the following example shows:

```
DECLARE
   TYPE PartTabTyp IS TABLE OF VARCHAR2(30)
      INDEX BY BINARY_INTEGER;
   part_tab   PartTabTyp;
   part_count INTEGER;
BEGIN
   part_tab(65) := 'OIL PAN';
   part_tab(97) := 'TRUNK LOCK';
   part_tab(44) := 'SHOCK ABSORBER';
   ...
   SELECT part_tab.COUNT    -- causes compilation error
      INTO part_count FROM dual;
   ...
END;
```

Using PL/SQL Tables

Mainly, you use PL/SQL tables to move bulk data into and out of database tables or between client–side applications and stored subprograms.

Retrieving Oracle Data

You can retrieve Oracle data into a PL/SQL table in three ways: the SELECT INTO statement lets you select a single row of data; the FETCH statement or a cursor FOR loop lets you fetch multiple rows.

Using the SELECT INTO statement, you can select a column entry into a scalar element. Or, you can select an entire row into a record element. In the following example, you select a row from the database table *dep*t into a record stored by the first element of the PL/SQL table *dept_tab*:

```
DECLARE
   TYPE DeptTabTyp IS TABLE OF dept%ROWTYPE
      INDEX BY BINARY_INTEGER;
   dept_tab DeptTabTyp;
BEGIN
   /* Select entire row into record stored by first element. */
   SELECT * INTO dept_tab(1) FROM dept WHERE deptno = 10;
   IF dept_tab(1).dname = 'ACCOUNTING' THEN ...
   ...
END;
```

Using the FETCH statement, you can fetch an entire column of Oracle data into a PL/SQL table of scalars. Or, you can fetch an entire table of Oracle data into a PL/SQL table of records. In the following example, you fetch rows from a cursor into the PL/SQL table of records *emp_tab*:

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab EmpTabTyp;
   i BINARY_INTEGER := 0;
   CURSOR c1 IS SELECT * FROM emp;
BEGIN
   OPEN c1;
   LOOP
      i := i + 1;
      /* Fetch entire row into record stored by ith element. */
      FETCH c1 INTO emp_tab(i);
      EXIT WHEN c1%NOTFOUND;
      -- process data record
   END LOOP;
   CLOSE c1;
END;
```

After loading PL/SQL tables of records this way, you can use them to simulate local database tables.

Instead of the FETCH statement, you can use a cursor FOR loop, which implicitly declares its loop index as a record, opens the cursor associated with a given query, repeatedly fetches rows of values into fields in the record, then closes the cursor. In the following example, you use a cursor FOR loop to fetch entire columns of Oracle data into the PL/SQL tables *ename_tab* and *sal_tab*:

```
DECLARE
   TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
      INDEX BY BINARY_INTEGER;
   TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
       INDEX BY BINARY_INTEGER;
   ename_tab EnameTabTyp;
   sal_tab   SalTabTyp;
   n BINARY_INTEGER := 0;
BEGIN
   /* Fetch entire columns into PL/SQL tables. */
   FOR emp_rec IN (SELECT ename, sal FROM emp) LOOP
      n := n + 1;
      ename_tab(n) := emp_rec.ename;
      sal_tab(n) := emp_rec.sal;
   END LOOP;
   ...
END;
```

Alternatively, you can place the cursor FOR loop in a standalone procedure. For example, given the declaration

```
CREATE PACKAGE emp_defs AS
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   ...
END emp_defs;
```

you might use the following standalone procedure to fetch all rows from the database table *emp* into the PL/SQL table of records *emp_tab:*

```
CREATE PROCEDURE load_emp_tab (
   n IN OUT BINARY_INTEGER,
   emp_tab OUT emp_defs.EmpTabTyp) AS  -- use packaged type
BEGIN
   n := 0;
   /* Fetch entire database table into PL/SQL table of records. */
   FOR emp_rec IN (SELECT * FROM emp) LOOP
      n := n + 1;
      emp_tab(n) := emp_rec;  -- assign record to nth element
   END LOOP;
END;
```

You can also use a cursor FOR loop to fetch Oracle data into packaged PL/SQL tables. For instance, given the declarations

```
CREATE PACKAGE emp_defs AS
   TYPE EmpnoTabTyp IS TABLE OF emp.empno%TYPE
      INDEX BY BINARY_INTEGER;
   empno_tab EmpnoTabTyp;
   ...
END emp_defs;
```

you might use the following block to fetch the database column *empno* into the public PL/SQL table *empno_tab:*

```
DECLARE
   ...
   i BINARY_INTEGER := 0;
BEGIN
   /* Fetch entire column into public PL/SQL table. */
   FOR emp_rec IN (SELECT empno FROM emp ORDER BY empno) LOOP
      i := i + 1;
      emp_defs.empno_tab(i) := emp_rec.empno;
   END LOOP;
   ...
END;
```

Inserting Oracle Data

You must use a loop to insert values from a PL/SQL table into a database column. For example, given the declarations

```
CREATE PACKAGE emp_defs AS
   TYPE EmpnoTabTyp IS TABLE OF emp.empno%TYPE
      INDEX BY BINARY_INTEGER;
   TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
      INDEX BY BINARY_INTEGER;
   empno_tab EmpnoTabTyp;
   ename_tab EnameTabTyp;
   ...
END emp_defs;
```

you might use the following standalone procedure to insert values from the PL/SQL tables *empno_tab* and *ename_tab* into the database table *emp*:

```
CREATE PROCEDURE insert_emp_ids (
   rows      IN BINARY_INTEGER,
   empno_tab IN EmpnoTabTyp,
   ename_tab IN EnameTabTyp) AS
BEGIN
   FOR i IN 1..rows LOOP
      INSERT INTO emp (empno, ename)
         VALUES (empno_tab(i), ename_tab(i));
   END LOOP;
END;
```

Restriction

You cannot reference record variables in the VALUES clause. So, you cannot insert entire records from a PL/SQL table of records into rows in a database table. For example, the following INSERT statement is illegal:

```
DECLARE
   TYPE DeptTabTyp IS TABLE OF dept%ROWTYPE
      INDEX BY BINARY_INTEGER;
   dept_tab DeptTabTyp;
   ...
BEGIN
   ...
   FOR i IN dept_tab.FIRST .. dept_tab.LAST LOOP
      INSERT INTO dept VALUES (dept_tab(i));  -- illegal
   END LOOP;
END;
```

Instead, you must specify one or more fields in the record, as the following example shows:

```
FOR i IN dept_tab.FIRST .. dept_tab.LAST LOOP
   INSERT INTO dept (deptno, dname)
      VALUES (dept_tab(i).deptno, dept_tab(i).dname);
END LOOP;
```

Using Host Arrays with PL/SQL Tables

With the Oracle Call Interface or the Oracle Precompilers, you can bind host arrays of scalars (but not host arrays of structures) to PL / SQL tables declared as the formal parameters of a subprogram. That allows you to pass host arrays to stored functions and procedures.

You can use a BINARY_INTEGER variable or compatible host variable to index the host arrays. Given the array subscript range *m* .. *n*, the corresponding PL / SQL table index range is always 1 .. *n* – *m* + 1. For example, if the array subscript range is 5 .. 10, the corresponding PL / SQL table index range is 1 .. (10 – 5 + 1) or 1 .. 6.

To assign all the values in a host array to elements in a PL / SQL table, you can use a subprogram call. In the Pro*C example below, you pass the host array *salary* to a PL / SQL block. From the block, you call a local function that declares the PL / SQL table *sal_tab* as one of its formal parameters. The function call assigns all values in the actual parameter *salary* to elements in the formal parameter *sal_tab*.

```
#include <stdio.h>
main()
{
   EXEC SQL BEGIN DECLARE SECTION;
      ...
      /* Declare host array. */
      float salary [100];
   EXEC SQL END DECLARE SECTION;

   /* Populate host array. */
   ...
   EXEC SQL EXECUTE
   DECLARE
      TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
         INDEX BY BINARY_INTEGER;
      mid_salary  REAL;
      n  BINARY_INTEGER := 100;
      FUNCTION median (sal_tab SalTabTyp, n INTEGER)
         RETURN REAL IS
      BEGIN
          -- compute median salary
      END;
   BEGIN
      mid_salary := median(:salary, n);  -- pass array
      ...
   END;
   END-EXEC;
   ...
}
```

Conversely, you can use a subprogram call to assign all values in a PL/SQL table to corresponding elements in a host array. In the Pro*C example below, you call a standalone procedure (not shown), which declares three PL/SQL tables as OUT formal parameters. The corresponding actual parameters are host arrays. When the procedure finishes fetching a batch of employee data into the PL/SQL tables, it assigns all values in the PL/SQL tables to elements in the host arrays.

```
#include <stdio.h>
...
EXEC SQL BEGIN DECLARE SECTION;
   ...
   int   array_size;
   int   number_returned;
   int   finished;
   /* Declare host arrays. */
   char  emp_name[10][11];
   char  job_title[10][10];
   float salary[10];
EXEC SQL END DECLARE SECTION;
...
main()
{
   ...
   array_size = 10;     /* determines batch size */
   number_returned = 0; /* needed for last batch */
   finished = 0;

   /* Array fetch loop. */
   for (;;)
   {
      EXEC SQL EXECUTE
      BEGIN
         /* Call stored procedure to fetch a batch of data. */
         get_emps(:emp_name, :job_title, :salary,
            :array_size, :number_returned, :finished);
      END;
      END-EXEC;

      print_rows(number_returned);

      if (finished) break;
   }
   ...
}
```

Table 4 – 1 shows the legal datatype conversions between row values in a PL / SQL table and elements in a host array. For example, a host array of type VARCHAR2 is compatible with a PL / SQL table of type LONG, LONG RAW, RAW, or VARCHAR2.

	PL/SQL Table							
Host Array	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	✓							
CHARZ	✓							
DATE		✓						
DECIMAL					✓			
DISPLAY					✓			
FLOAT					✓			
INTEGER					✓			
LONG	✓		✓					
LONG VARCHAR			✓	✓		✓		✓
LONG VARRAW				✓		✓		
NUMBER					✓			
RAW				✓		✓		
ROWID							✓	
STRING			✓	✓		✓		✓
UNSIGNED					✓			
VARCHAR			✓	✓		✓		✓
VARCHAR2			✓	✓		✓		✓
VARNUM					✓			
VARRAW				✓		✓		

Table 4 – 1 Legal Datatype Conversions

ARRAYLEN Statement

Suppose you pass a host array to a PL/SQL block for processing. By default, when binding the host array, the Oracle Precompilers use its declared dimension. However, you might not want to process the entire array, in which case you can use the ARRAYLEN statement to specify a smaller dimension. ARRAYLEN associates the host array with a host variable, which stores the smaller dimension.

Let us repeat the first example above using ARRAYLEN to override the default dimension of the host array *salary*:

```
#include <stdio.h>
main()
{
   EXEC SQL BEGIN DECLARE SECTION;
      ...
      /* Declare host array. */
      float salary [100];
      int   my_dim;
      EXEC SQL ARRAYLEN salary (my_dim);
   EXEC SQL END DECLARE SECTION;
   /* Populate host array. */
   ...
   /* Set smaller host array dimension. */
   my_dim = 25;
   EXEC SQL EXECUTE
   DECLARE
      TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
         INDEX BY BINARY_INTEGER;
      mid_salary  REAL;
      FUNCTION median (sal_tab SalTabTyp, n INTEGER)
         RETURN REAL IS
      BEGIN
         ... -- compute median salary
      END;
   BEGIN
      mid_salary := median(:salary, :my_dim);  -- pass array
      ...
   END;
   END-EXEC;
   ...
}
```

Only 25 array elements are passed to the PL/SQL block because ARRAYLEN downsizes the host array from 100 to 25 elements. As a result, when the PL/SQL block is sent to Oracle for execution, a much smaller host array is sent along. This saves time and reduces network traffic.

User–Defined 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, you cannot specify the datatypes of fields in the record or declare fields of your own. The composite datatype RECORD lifts those restrictions.

As you might expect, objects of type RECORD are called *records*. 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.

Defining RECORD Types

Records must be declared in two steps. First, you define a RECORD type, then declare user–defined records of that type. You can define RECORD types in the declarative part of any block, subprogram, or package using the syntax

```
TYPE record_type_name IS RECORD (field[, field]...);
```

where *record_type_name* is a type specifier used in subsequent declarations of records and *field* stands for the following syntax:

```
field_name datatype [[NOT NULL] {:= | DEFAULT} expr]
```

You can use the attributes %TYPE and %ROWTYPE to specify field types. In the following example, you define a RECORD type named *DeptRecTyp*:

```
DECLARE
   TYPE DeptRecTyp IS RECORD (
      deptno NUMBER(2),
      dname  dept.dname%TYPE,
      loc    dept.loc%TYPE);
```

Notice that the field declarations are like variable declarations. Each field has a unique name and specific datatype.

The next example shows that you can initialize a RECORD type. When you declare a record of type *TimeTyp*, its three fields assume an initial value of zero.

```
DECLARE
   TYPE TimeTyp IS RECORD (
      seconds SMALLINT := 0,
      minutes SMALLINT := 0,
      hours   SMALLINT := 0);
```

You can add the NOT NULL constraint to any field declaration and so prevent the assigning of nulls to that field. Fields declared as NOT NULL must be initialized.

Nested Records

PL/SQL lets you define *nested* records. That is, a record can be the component of another record, as the following example shows:

```
DECLARE
   TYPE TimeTyp IS RECORD (
      seconds SMALLINT,
      minutes SMALLINT,
      hours   SMALLINT);
   TYPE MeetingTyp IS RECORD (
      day     DATE,
      time    TimeTyp,  -- nested record
      place   VARCHAR2(20),
      purpose VARCHAR2(50));
```

Function Results

The example below 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 EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
   ...
   FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS ...
```

Declaring Records

Once you define a RECORD type, you can declare records of that type, as the following example shows:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      emp_id    NUMBER(4),
      emp_name  CHAR(10),
      job_title CHAR(9)
      hire_date DATE));
   emp_rec EmpRecTyp;  -- declare user-defined record
```

The identifier *emp_rec* represents an entire record.

Like scalar variables, user–defined records can be declared as the formal parameters of procedures and functions. An example follows:

```
CREATE PACKAGE emp_actions AS
   TYPE EmpRecTyp IS RECORD (
      emp_id    NUMBER(4),
      last_name CHAR(10),
      job_title CHAR(14), ...);
   ...
   PROCEDURE hire_employee (emp_rec EmpRecTyp);
```

Referencing Records

To reference individual fields in a record, you use dot notation and the following syntax:

```
record_name.field_name
```

For example, you reference the field *hire_date* in the record *emp_rec* as follows:

```
emp_rec.hire_date ...
```

You can assign the value of an expression to a specific field using the following syntax:

```
record_name.field_name := expression;
```

In the next example, you convert an employee name to upper case:

```
emp_rec.ename := UPPER(emp_rec.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 record to another if they have the same datatype, as the following example shows:

```
DECLARE
   TYPE DeptRecTyp IS RECORD(...);
   dept_rec1 DeptRecTyp;
   dept_rec2 DeptRecTyp;
BEGIN
   ...
   dept_rec1 := dept_rec2;  -- assign one record to another
```

Records that have different datatypes cannot be assigned to each other even if their fields match exactly.

Note: A user–defined record and a %ROWTYPE record always have different datatypes.

Second, you can use the SELECT or FETCH statement to fetch column values into a record, as the example below shows. The column names must appear in the same order as the fields in your record.

```
DECLARE
   TYPE DeptRecTyp IS RECORD(
      dept_no   NUMBER(2),
      dept_name CHAR(14),
      location  CHAR(13));
   dept_rec DeptRecTyp;
BEGIN
   SELECT deptno, dname, loc INTO dept_rec FROM dept WHERE ...
```

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_rec);  -- 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
```

Finally, records cannot be tested for equality, inequality, or nullity. For instance, the following IF conditions are illegal:

```
IF dept_rec1 = dept_rec2 THEN ...  -- illegal
IF emp_rec IS NULL THEN ...  -- illegal
```

Nested Records

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 parent records have different datatypes.

```
DECLARE
   TYPE TimeTyp IS RECORD (minutes SMALLINT, hours SMALLINT);
   TYPE MeetingTyp IS RECORD (
      day     DATE,
      time    TimeTyp,  -- nested record
      room    INTEGER(4),
      subject VARCHAR2(35));
   TYPE PartyTyp IS RECORD (
      day   DATE,
      time  TimeTyp,  -- nested record
      place VARCHAR2(15));
   meeting MeetingTyp;
   seminar MeetingTyp;
   party   PartyTyp;
   ...
BEGIN
   ...
   seminar.time := meeting.time;  -- same parent type
   party.time := meeting.time;  -- different parent types
   ...
END;
```

Function Results

When calling a function that returns a user–defined record, you use the following syntax to reference fields in the record:

```
function_name(parameters).field_name
```

For example, the following call to the function *nth_highest_sal* references the field *salary* in the user–defined record *emp_rec*:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      emp_id    NUMBER(4),
      job_title CHAR(14),
      salary    REAL);
   middle_sal REAL;
   FUNCTION nth_highest_sal (n INTEGER) RETURN EmpRecTyp IS
      emp_rec EmpRecTyp;
   BEGIN
      ...
      RETURN emp_rec;  -- return user-defined record
   END;
BEGIN
   ...
   middle_sal := nth_highest_sal(10).salary;  -- call function
```

To reference nested fields in a record returned by a function, you use the following syntax:

```
function_name(parameters).field_name.nested_field_name
```

For example, the following call to the function *calendar_item* references the nested field *hours* in the user–defined record *meeting*:

```
DECLARE
   TYPE TimeTyp IS RECORD (minutes SMALLINT, hours SMALLINT);
   TYPE MeetingTyp IS RECORD (
      day      DATE,
      duration TimeTyp,  -- nested record
      room     INTEGER(4),
      subject  VARCHAR2(35));
   ...
   FUNCTION calendar_item (priority INTEGER) RETURN MeetingTyp IS
      meeting MeetingTyp;
   BEGIN
      ...
      RETURN meeting;  -- return user-defined record
   END;
BEGIN
   ...
   IF calendar_item(3).duration.hours > 2 THEN ...
```

Restriction

Currently, you cannot use the syntax above to call a parameterless function because PL/SQL does not allow empty parameter lists. That is, the following syntax is illegal:

```
function_name().field_name  -- illegal; empty parameter list
```

You cannot just drop the empty parameter list because the following syntax is also illegal:

```
function_name.field_name  -- illegal; no parameter list
```

Instead, declare a local user–defined record to which you can assign the function result, then reference its fields directly, as shown in the following example:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (..., salary REAL);
   emp_rec EmpRecTyp;  -- declare record
   median  REAL;
   FUNCTION median_sal RETURN EmpRecTyp IS ...
BEGIN
   ...
   emp_rec := median_sal;  -- assign function result
   median := emp_rec.salary;  -- reference field
```

Using Records

The RECORD type 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 the database tables *assets* and *liabilities*, then use ratio analysis to compare the performance of two subsidiary companies:

```
DECLARE
   TYPE FiguresTyp IS RECORD (cash REAL, notes REAL, ...);
   sub1_figs FiguresTyp;
   sub2_figs FiguresTyp;
   ...
   FUNCTION acid_test (figs FiguresTyp) 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.

CHAPTER

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.

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 supports all SQL data manipulation commands (except EXPLAIN PLAN), transaction control commands, functions, pseudocolumns, and operators. Also, PL/SQL conforms to SQL92, the current ANSI/ISO SQL standard.

Note: PL/SQL does *not* support data definition commands such as ALTER and CREATE. For an explanation and workaround, see "Using DDL and Dynamic SQL" on page 5 – 7.

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 establishes a read–only transaction.

SQL Functions

PL/SQL lets you use all the SQL functions including the following group functions, which summarize entire columns of Oracle data: AVG, COUNT, MAX, MIN, STDDEV, SUM, and VARIANCE.

The group functions GLB and LUB are available only with Trusted Oracle. GLB and LUB return the greatest lower bound and least upper bound of an operating system label, respectively. For more information, see *Trusted Oracle7 Server Administrator's Guide*.

You can use the group functions in SQL statements, but *not* in procedural statements. Group 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 group function treats all returned rows as a single group.

You call a group function using the syntax

```
function_name([ALL | DISTINCT] expr)
```

where *expr* is an expression that refers to one or more database columns. If you specify the ALL option (the default), the group function considers all column values including duplicates. For example, the following statement returns the sample standard deviation (s) of all values in the *comm* column:

```
SELECT STDDEV(comm) INTO comm_sigma FROM emp;
```

If you specify the DISTINCT option, the group function considers only distinct values. For example, the following statement returns the number of different job titles in the *emp* table:

```
SELECT COUNT(DISTINCT job) INTO job_count FROM emp;
```

The COUNT function lets you specify the asterisk (*) option, which returns the number of rows in a table. For example, the following statement returns the number of employees in the *emp* table:

```
SELECT COUNT(*) INTO emp_count FROM emp;
```

Except for COUNT(*), all group functions ignore nulls.

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 values into, update values in, or delete values from a pseudocolumn.

You can use pseudocolumns in SQL statements, but *not* in procedural statements. In the following example, you use the database sequence *empno_seq* and the pseudocolumn NEXTVAL (which returns the next value in a database sequence) to insert a new employee number into the *emp* table:

```
INSERT INTO emp VALUES (empno_seq.NEXTVAL, new_ename, ...);
```

Brief descriptions of the pseudocolumns follow. For more information, see *Oracle7 Server SQL Reference.*

CURRVAL and NEXTVAL

A *sequence* is a database 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. Recall that PL/SQL provides a datatype also named ROWID. You can use variables of type ROWID to store rowids in a readable format. In the following example, you declare a variable named *row_id* for that purpose:

```
DECLARE
   row_id ROWID;
```

When you select or fetch a rowid into a ROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to an 18–byte character string. Then, you can compare the ROWID 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 – 47.

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;
```

For example, the following condition cannot be met because the first nine rows are never retrieved:

```
... WHERE ROWNUM = 10;
```

ROWLABEL Column

PL/SQL also recognizes the special column ROWLABEL, which Trusted Oracle creates for every database table. Like other columns, ROWLABEL can be referenced in SQL statements. However, with standard Oracle, ROWLABEL returns a null. With Trusted Oracle, ROWLABEL returns the operating system label for a row.

A common use of ROWLABEL is to filter query results. For example, the following statement counts only those rows with a security level higher than "unclassified":

```
SELECT COUNT(*) INTO head_count FROM emp
   WHERE ROWLABEL > 'UNCLASSIFIED';
```

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 *Oracle7 Server 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 yields 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.

ALL	Compares a value to each value in a list or returned by a subquery and yields TRUE if all of the individual comparisons yield TRUE.
ANY, SOME	Compares a value to each value in a list or returned by a subquery and yields TRUE if any of the individual comparisons yields TRUE.
BETWEEN	Tests whether a value lies in a specified range.
EXISTS	Returns TRUE if a subquery returns at least one row.
IN	Tests for set membership.
IS NULL	Tests for nulls.
LIKE	Tests whether a character string matches a specified pattern, which can include wildcards.

Set Operators

Set operators combine the results of two queries into one result. INTERSECT returns all distinct rows selected by both queries. MINUS returns all distinct rows selected by the first query but not by the second. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

Row Operators

Row operators return or reference particular rows. ALL retains duplicate rows in the result of a query or in an aggregate expression. DISTINCT eliminates duplicate rows from the result of a query or from an aggregate expression. PRIOR refers to the parent row of the current row returned by a tree–structured query. You must use this operator in the CONNECT BY clause of such a query to define the parent–child relationship.

SQL92 Conformance

In late 1992, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) adopted the current SQL standard known informally as SQL92, which greatly extends the previous SQL standard (SQL89).

Note: SQL92 is known officially as International Standard ISO / IEC 9075:1992, *Database Language SQL*, which is also available as ANSI Document ANSI X3.135–1992.

SQL92 specifies a "conforming SQL language" and, to allow implementation in stages, defines three language levels:

- Full SQL
- Intermediate SQL (a subset of Full SQL)
- Entry SQL (a subset of Intermediate SQL)

A conforming SQL implementation must support at least Entry SQL. PL / SQL fully supports Entry SQL.

Using DDL and Dynamic SQL

This section explains why PL / SQL does not support SQL data definition language (DDL) or dynamic SQL, then shows how to solve the problem.

Efficiency versus Flexibility

Before a PL / SQL program can be executed, it must be compiled. The PL / SQL compiler resolves references to Oracle objects by looking up their definitions in the data dictionary. Then, the compiler assigns storage addresses to program variables that will hold Oracle data so that Oracle can look up the addresses at run time. This process is called *binding*.

How a database language implements binding affects runtime efficiency and flexibility. Binding at compile time, called *static* or *early* binding, increases efficiency because the definitions of database objects are looked up then, not at run time. On the other hand, binding at run time, called *dynamic* or *late* binding, increases flexibility because the definitions of database objects can remain unknown until then.

Designed primarily for high–speed transaction processing, PL / SQL increases efficiency by bundling SQL statements and avoiding runtime compilation. Unlike SQL, which is compiled and executed statement–by–statement at run time (late binding), PL / SQL is processed into machine–readable p–code at compile time (early binding). At run time, the PL / SQL engine simply executes the p–code.

Some Limitations

However, this design imposes some limitations. For example, the p–code includes references to database objects such as tables and stored procedures. The PL/SQL compiler can resolve such references only if the database objects are known at compile time. In the following example, the compiler cannot process the procedure because the table is undefined until the procedure is executed at run time:

```
CREATE PROCEDURE create_table AS
BEGIN
   CREATE TABLE dept (deptno NUMBER(2), ...);  -- illegal
   ...
END;
```

In the next example, the compiler cannot bind the table reference in the DROP TABLE statement because the table name is unknown until the procedure is executed:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
   DROP TABLE table_name;  -- illegal
   ...
END;
```

Overcoming the Limitations

However, the package DBMS_SQL, which is supplied with Oracle7, allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time. For example, when called, the following stored procedure drops a specified database table:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
   cid INTEGER;
BEGIN
   /* Open new cursor and return cursor ID. */
   cid := DBMS_SQL.OPEN_CURSOR;
   /* Parse and immediately execute dynamic SQL statement built by
      concatenating table name to DROP TABLE command. */
   DBMS_SQL.PARSE(cid, 'DROP TABLE ' || table_name, dbms_sql.v7);
   /* Close cursor. */
   DBMS_SQL.CLOSE_CURSOR(cid);
EXCEPTION
   /* If an exception is raised, close cursor before exiting. */
   WHEN OTHERS THEN
      DBMS_SQL.CLOSE_CURSOR(cid);
      RAISE;  -- reraise the exception
END drop_table;
```

For more information about package DBMS_SQL, see *Oracle7 Server Application Developer's Guide.*

Managing Cursors

Recall from Chapter 1 that 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. 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]...)]
   IS select_statement;
```

where *parameter* stands for the following syntax:

```
cursor_parameter_name [IN] datatype [{:= | DEFAULT} expr]
```

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 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 NOT NULL constraint 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 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 – 12.) 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 of the FETCH statement 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 variable in the INTO list. Also, their datatypes must be compatible. Typically, you use the FETCH statement as follows:

```
OPEN c1;
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 each 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 – 33.

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

In this context, a *subquery* is a query that appears in another query. When evaluated, the subquery provides a value or set of values to the query. Subqueries are most often 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');
         FROM emp GROUP BY deptno) t2
      WHERE t1.deptno = t2.deptno AND "STAFF" => 5;
```

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 – 33.

Packaging Cursors

You can separate a cursor specification from its body for placement in a package. That way, you can change the cursor body without having to change the cursor specification. You code the cursor specification in the package specification using the syntax

```
CURSOR cursor_name [(parameter[, parameter]...)]
    RETURN return_type;
```

where *return_type* must represent a record or a row in a database table. 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 specification. */
   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 specification 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 specification. Also, the number and datatypes of select items in the SELECT statement must match the RETURN clause.

Packaged cursors increase flexibility. For instance, you can change the cursor body in the last example, as follows, without having to change the cursor specification:

```
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 explicitly declared 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. This is true even if you use an EXIT or GOTO statement to leave the loop prematurely or if an exception is raised inside the loop.

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 query 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 next 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, you 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 used 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 how many have 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 subprograms, including subprograms stored in an Oracle database. 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 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 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 object instead of the object itself. So, declaring a cursor variable creates a pointer, *not* an object. 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. Currently, cursor variables are the only REF variables that you can declare.

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 package variables, cursor variables do not have persistent state. Remember, declaring a cursor variable creates a pointer, not an object. 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 (previously declared) 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
```

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 ...
```

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. The statement syntax is

```
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 do not take parameters. No flexibility is lost, however, because you can pass whole queries (not just parameters) to a cursor variable. The query can reference bind 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.

In a Stored Procedure

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 standalone procedure to open the cursor variable. Simply define the REF CURSOR type in a separate package, then reference that type in the standalone procedure. For instance, if you create the following (bodiless) package, you can create standalone procedures that reference the types it defines:

```
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 following example, you create a standalone 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 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 and selector to a stored procedure that executes queries with different return types. Consider the following example:

```
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;
```

Using a Bind 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 bind 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. The statement syntax follows:

```
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. The statement syntax follows:

```
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.

Some Examples

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 (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;
```

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;
```

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. */
}
```

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 five 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 Exceptions

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 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.

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 standalone procedure that references the new type. Then, inside a PL/SQL block, you open a host cursor variable for a query of the *dept* table. Later, when you pass the open host cursor variable to the stored procedure, PL/SQL raises ROWTYPE_MISMATCH because the return types of the actual and formal parameters are incompatible.

```
/* bodiless package */
CREATE PACKAGE cv_types AS
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   ...
END cv_types;

/* standalone procedure */
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 because emp and
                        -- dept tables have different rowtypes
   END;
END-EXEC;
```

Guarding Against Aliasing

Like all pointers, cursor variables introduce 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 get_emp_data;
```

Aliasing also occurs when the same actual parameter appears twice in a subprogram call. Unless both formal parameters are IN parameters, the result is indeterminate, as the following example shows:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   emp_cv  EmpCurTyp;
   emp_rec emp%ROWTYPE;
   PROCEDURE open_emp_cv (cv1 IN OUT EmpCurTyp,
                          cv2 IN OUT EmpCurTyp) IS
   BEGIN
      OPEN cv1 FOR SELECT * FROM emp WHERE ename = 'KING';
      OPEN cv2 FOR SELECT * FROM emp WHERE ename = 'BLACK';
   END open_emp_cv;
BEGIN
   open_emp_cv(emp_cv, emp_cv);
   FETCH emp_cv INTO emp_rec;  -- indeterminate; might return
                               -- row for 'KING' or 'BLACK'
   ...
END;
```

Restrictions

Currently, cursor variables are subject to the following restrictions, some of which future releases of PL/SQL will remove:

- You cannot declare cursor variables in a package because they do not have persistent state.
- 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 (bind 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. For example, the following IF conditions are illegal:

```
PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp,
                       tmp_cv IN OUT TmpCurTyp) IS
BEGIN
   ...
   IF emp_cv = tmp_cv THEN ...  -- illegal
   IF emp_cv IS NULL THEN ...  -- illegal
END;
```

- You cannot assign nulls to a cursor variable. For example, the following assignment statement is illegal:

```
PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
BEGIN
   emp_cv := NULL;  -- illegal
   ...
END;
```

- 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 PL/SQL table, which means that elements in a PL/SQL table cannot store the values of cursor variables. For instance, the following TABLE type definition is illegal:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   TYPE EmpCurTabTyp IS TABLE OF EmpCurTyp  -- illegal
      INDEX BY BINARY_INTEGER;
```

- 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 LOOP;
END;
```

- You cannot use cursor variables with dynamic SQL.

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;
```

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.

		%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
middle FETCHes	**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.

Table 5 – 1 Cursor Attribute Values

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;  -- yields 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, ...);
   ...
END IF;
```

%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. In this example, you use %NOTFOUND to insert a new row if an update fails:

```
UPDATE emp SET sal = sal * 1.05 WHERE empno = my_empno;
IF SQL%NOTFOUND THEN  -- update failed
   INSERT INTO errors VALUES (...);
END IF;
```

%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:

```
UPDATE parts SET quantity = quantity – 1 WHERE partno = part_id;
check_status(part_id);  –– procedure call
IF SQL%NOTFOUND THEN  –– dangerous!
```

You can debug the code as follows:

```
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 ...
```

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 group function never raises NO_DATA_FOUND because group 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. Before delving into the subject, you should know the terms defined below.

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 or structures 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 database 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 information in the rollback segment to build read–consistent query results and to undo changes if necessary.

How Transactions Guard Your Database

Oracle is transaction oriented; that is, it uses transactions to ensure data integrity. A transaction is a series of one or more logically related SQL statements that accomplish a task. 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 database 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 rollback 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;
```

By default, the number of active savepoints per session is limited to five. An *active savepoint* is one marked since the last commit or rollback. You or your DBA can raise the limit (up to 255) by increasing the value of the Oracle initialization parameter SAVEPOINTS.

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

It is good programming practice 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;
```

In the OCI environment, if you issue the OLOGOF call, Oracle automatically commits your transaction. Otherwise, Oracle rolls back the transaction.

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

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 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.

You can explicitly lock entire tables using the LOCK TABLE statement.

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. The wait has no limit unless the table is remote, in which case the Oracle initialization parameter DISTRIBUTED_LOCK_TIMEOUT sets a limit.

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 – 47.)

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;
```

You use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row fetched from a cursor, as the following example shows:

```
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 *Oracle7 Server Application Developer's Guide*.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row will one transaction wait for the other to complete.

Note: If your program includes SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks.

Fetching Across Commits

Remember, 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 ROWID 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  ROWID;
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 primarily for transaction processing. As a result, the PL/SQL compiler limits the number of tokens a block can generate. Blocks that exceed the limit cause a *program too large* compilation error. Generally, blocks larger than 64K exceed the token limit. However, much smaller blocks can exceed the limit if they contain many variables or complex SQL statements.

The best solution to this problem is to modularize your program by defining subprograms (which can be stored in an Oracle database). For more information, see Chapter 7.

Another solution is to break the block into two sub–blocks. Before the first block terminates, have it insert any data the second block needs into a database table called *temp* (for example). When the second block starts executing, have it select the data from *temp*. This approximates the passing of parameters from one procedure to another. The following example shows two “parameter passing” PL/SQL blocks in a SQL*Plus script:

```
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;
/
```

The previous 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. To avoid these restrictions, embed your PL/SQL blocks in a third–generation host language such as C, COBOL, or FORTRAN. 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 using a temporary database table. In the following example, you embed two PL/SQL blocks in a Pro*C program:

```
EXEC SQL BEGIN DECLARE SECTION;
   int   my_empno;
   float my_sal, my_comm;
   short comm_ind;
   ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL EXECUTE
   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;
END-EXEC;
...
EXEC SQL EXECUTE
   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;
END-EXEC;
...
```

CHAPTER

6 Error Handling

There is nothing more exhilarating than to be shot at without result.

Winston Churchill

Runtime 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 runtime 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.

Overview

In PL/SQL, a warning or error condition is called an *exception*. Exceptions can be internally defined (by the runtime system) or user defined. Examples of internally defined exceptions include *division by zero* and *out of memory*. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named *insufficient_funds* to flag overdrawn bank accounts. Unlike internal exceptions, user–defined exceptions *must* be given names.

When an error occurs, an exception is *raised*. That is, normal execution stops and control transfers to the exception–handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the runtime 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 and Disadvantages 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, as follows:

```
BEGIN
   SELECT ...
      -- check for 'no data found' error
   SELECT ...
      -- check for 'no data found' error
   SELECT ...
      -- check for 'no data found' error
   ...
END;
```

Error processing is not clearly separated from normal processing; nor is it robust. If you neglect to code a check, the error goes undetected and is likely to cause other, seemingly unrelated errors.

With exceptions, you can handle errors conveniently without the need to code multiple checks, as follows:

```
BEGIN
   SELECT ...
   SELECT ...
   SELECT ...
   ...
EXCEPTION
   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
      ...
END;
```

Notice how 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.

Disadvantages

Using exceptions for error handling has two disadvantages. First, exceptions can trap only runtime errors. Therefore, a PL/SQL program cannot trap and recover from compile–time (syntax and semantic) errors such as *table or view does not exist*.

Second, 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;
```

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 error–reporting 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 numbers. (See "Using EXCEPTION_INIT" on page 6 – 9.)

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 Name	*Oracle Error*	*SQLCODE Value*
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
STORAGE_ERROR	ORA–06500	–6500
TIMEOUT_ON_RESOURCE	ORA–00051	–51
TOO_MANY_ROWS	ORA–01422	–1422
VALUE_ERROR	ORA–06502	–6502
ZERO_DIVIDE	ORA–01476	–1476

For a complete list of the messages that Oracle or PL/SQL might issue, see *Oracle7 Server Messages*.

Brief descriptions of the predefined exceptions follow:

CURSOR_ALREADY_OPEN is raised if you try to open an already open cursor. You must close a cursor before you can reopen it.

A cursor FOR loop automatically opens the cursor to which it refers. Therefore, you cannot enter the loop if that cursor is already open, nor can you open that cursor inside the loop.

DUP_VAL_ON_INDEX is raised if you try to store duplicate values in a database column that is constrained by a unique index.

INVALID_CURSOR is raised if you try an illegal cursor operation. For example, INVALID_CURSOR is raised if you close an unopened cursor.

INVALID_NUMBER is raised in a SQL statement if the conversion of a character string to a number fails because the string does not represent a valid number. For example, the following INSERT statement raises INVALID_NUMBER when Oracle tries to convert 'HALL' to a number:

```
INSERT INTO emp (empno, ename, deptno) VALUES ('HALL', 7888, 20);
```

In procedural statements, VALUE_ERROR is raised instead.

LOGIN_DENIED is raised if you try logging on to Oracle with an invalid username/password.

NO_DATA_FOUND is raised if a SELECT INTO statement returns no rows or if you reference an uninitialized row in a PL/SQL table. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.

SQL group functions such as AVG and SUM *always* return a value or a null. So, a SELECT INTO statement that calls a group function will never raise NO_DATA_FOUND.

NOT_LOGGED_ON is raised if your PL/SQL program issues a database call without being connected to Oracle.

PROGRAM_ERROR is raised if PL/SQL has an internal problem.

ROWTYPE_MISMATCH is raised if the host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, when you pass an open host cursor variable to a stored subprogram, if the return types of the actual and formal parameters are incompatible, PL/SQL raises ROWTYPE_MISMATCH.

STORAGE_ERROR is raised if PL/SQL runs out of memory or if memory is corrupted.

TIMEOUT_ON_RESOURCE is raised if a timeout occurs while Oracle is waiting for a resource.

TOO_MANY_ROWS is raised if a SELECT INTO statement returns more than one row.

VALUE_ERROR is raised if an arithmetic, conversion, truncation, or size–constraint error occurs. For example, when you select 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 to a number fails. For example, the following assignment statement raises VALUE_ERROR when PL/SQL tries to convert 'HALL' to a number:

```
DECLARE
   my_empno NUMBER(4);
   my_ename CHAR(10);
BEGIN
   my_empno := 'HALL';  -- raises VALUE_ERROR
```

In SQL statements, INVALID_NUMBER is raised instead.

ZERO_DIVIDE is raised if you try to divide a number by zero because the result is undefined.

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;
   acct_num NUMBER(5);
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not an object. 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 next 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. They do not affect the meaning of a program; they simply convey information to the compiler. 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 part, as shown in the following example:

```
DECLARE
   insufficient_privileges EXCEPTION;
   PRAGMA EXCEPTION_INIT(insufficient_privileges, –1031);
      ------------------------------------------------------
      -- Oracle returns error number –1031 if, for example,
      -- you try to UPDATE a table for which you have
      -- only SELECT privileges
      ------------------------------------------------------
BEGIN
   ...
EXCEPTION
   WHEN insufficient_privileges THEN
      -- handle the error
   ...
END;
```

Using raise_application_error

Package DBMS_STANDARD, which is supplied with Oracle7, 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*, you 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 it.

An application can call *raise_application_error* only from an executing stored subprogram. 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, increase NUMBER) AS
   current_salary NUMBER;
BEGIN
   SELECT sal INTO current_salary FROM emp
      WHERE empno = emp_id;
   IF current_salary IS NULL THEN
      /* Issue user-defined error message. */
      raise_application_error(-20101, 'Salary is missing');
   ELSE
      UPDATE emp SET sal = current_salary + increase
         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
   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(:emp_number, :amount);
   EXCEPTION
      WHEN null_salary THEN
         INSERT INTO emp_audit VALUES (:emp_number, ...);
      ...
   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
   ...
   WHEN OTHERS THEN ...
END;
```

How Exceptions Are Raised

Internal exceptions are raised implicitly by the runtime 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 – 10.

Figure 6 – 1, Figure 6 – 2, and Figure 6 – 3 illustrate the basic propagation rules.

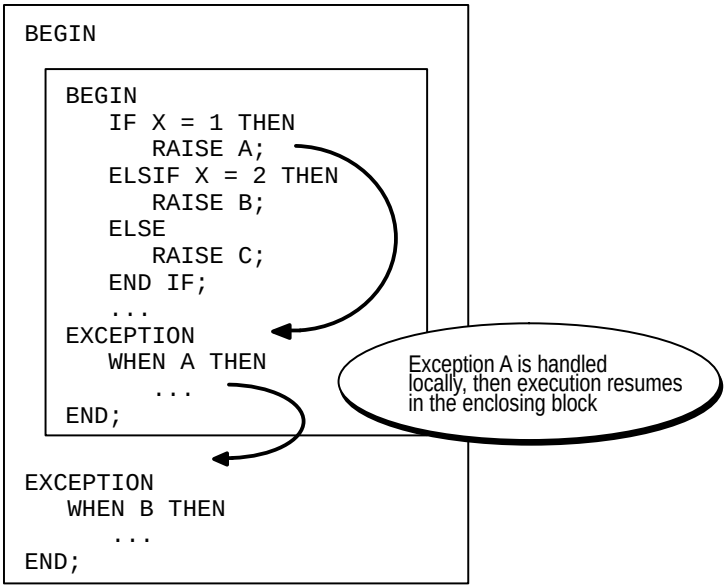

Figure 6 – 1 Propagation Rules: Example 1

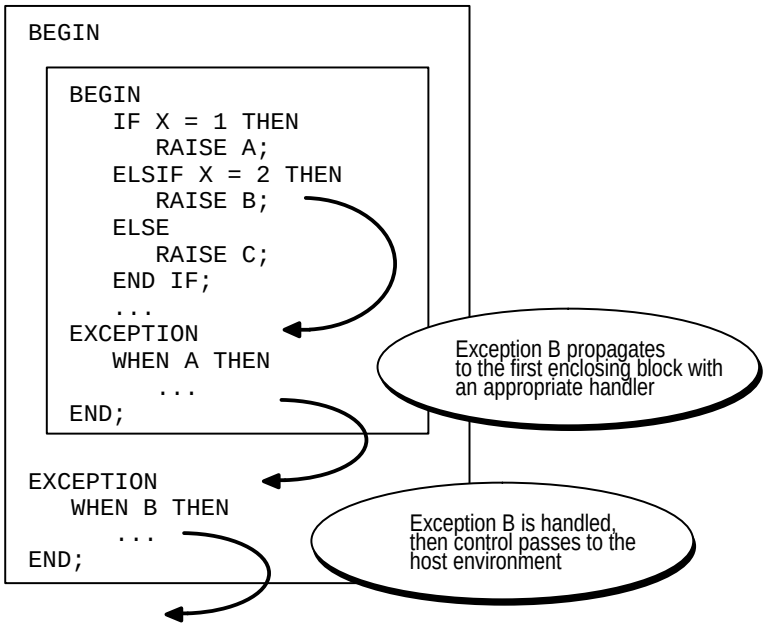

Figure 6 – 2 Propagation Rules: Example 2

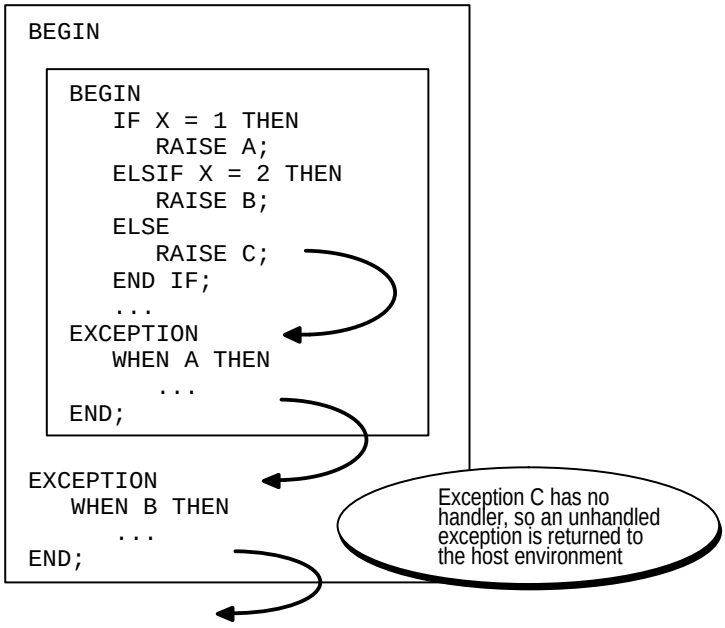

Figure 6 – 3 Propagation Rules: Example 3

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 it was declared has no handler for the exception named *past_due*, it 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 must 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 *limit* cannot store numbers larger than 999:

```
DECLARE
   limit CONSTANT NUMBER(3) := 5000;  -- raises an exception
BEGIN
   ...
EXCEPTION
   WHEN OTHERS THEN ...  -- cannot catch the exception
```

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 to an exception handler; nor can it 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 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 this message:

```
ORA–01403: no data found
```

Passing a zero to SQLERRM always returns the following message:

```
ORA–0000: normal, successful completion
```

You cannot use SQLCODE or SQLERRM directly in a SQL statement. For example, the following statement is illegal:

```
INSERT INTO errors VALUES (SQLCODE, SQLERRM);
```

Instead, you must assign their values to local variables, then use the variables in the SQL statement, as the following example shows:

```
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*. 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 block and subprogram.

Useful Techniques

In this section, you learn two useful techniques: how to continue after an exception is raised and how to retry a transaction.

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 ...
```

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 ...
```

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   CHAR(20);
   ans1   CHAR(3);
   ans2   CHAR(3);
   ans3   CHAR(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 (because there is a unique
            -- index on the name column)
         COMMIT;
         EXIT;
      EXCEPTION
         WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK TO start_transaction;  -- undo changes
            suffix := suffix + 1;             -- try to fix
            name := name || TO_CHAR(suffix);  -- problem
         ...
      END;  -- sub-block ends
   END LOOP;
END;
```

CHAPTER

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.

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 objects 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 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;
```

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 CHAR, new_loc CHAR) 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.

Also, subprograms promote *reusability* and *maintainability*. Once validated, a subprogram can be used with confidence in any number of applications. Furthermore, only the subprogram is affected if its definition changes. 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 | IN OUT] datatype [{:= | DEFAULT} expr]
```

You cannot impose the NOT NULL constraint on a parameter.

Also, you cannot specify a constraint on the datatype. For example, the following declaration of *emp_id* is illegal:

```
PROCEDURE ... (emp_id NUMBER(4)) IS  -- illegal; should be NUMBER
BEGIN ... END;
```

A procedure has two parts: the specification and the body. The procedure specification 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:

```
PROCEDURE raise_salary (emp_id INTEGER, increase 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 + increase
         WHERE empno = emp_id;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO emp_audit VALUES (emp_id, 'No such number');
   WHEN salary_missing THEN
      INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

When called, this procedure accepts an employee number and a salary increase amount. It uses the employee number to select the current salary from the *emp* database table. If the employee number is not found or if the current salary is null, an exception is raised. Otherwise, the salary is updated.

A procedure is called as a PL/SQL statement. For example, you might call the procedure *raise_salary* as follows:

```
raise_salary(emp_num, 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 | IN OUT] datatype [{:= | DEFAULT} expr]
```

Remember, you cannot impose the NOT NULL constraint on a parameter, and you cannot specify a constraint on the datatype.

Like a procedure, a function has two parts: the specification and the body. The function specification 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:

```
IF sal_ok(new_sal, new_title) THEN ...
```

The function identifier acts like a variable whose value depends on the parameters passed to it.

Restriction

To be callable from SQL expressions, a stored function must obey certain rules meant to control side effects. For standalone functions, Oracle can enforce these rules by checking the function body. However, the body of a packaged function is hidden. So, for packaged functions, you must use the pragma RESTRICT_REFERENCES to enforce the rules. For more information, see “Calling Stored Functions from SQL Expressions” in *Oracle7 Server Application Developer’s Guide.*

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 specification.)

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, it is poor programming practice to have multiple exit points in a subprogram.

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 acctno = 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. However, you must declare subprograms at the end of a declarative section after all other program objects. 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 this 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 – 23)
- group subprograms in a package

A forward declaration consists of a subprogram specification 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 specifications go in the package specification, 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 specification
   PROCEDURE hire_employee (emp_id INTGER, name VARCHAR2, ...);
   PROCEDURE fire_employee (emp_id INTEGER);
   PROCEDURE raise_salary (emp_id INTEGER, increase 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, increase 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 specifications in the package specification. 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 by all tools, subprograms must be stored in an Oracle database.

To create subprograms and store them permanently in an Oracle database, you use the CREATE PROCEDURE and CREATE FUNCTION statements, which you can execute interactively from SQL*Plus or Server Manager. For example, you might create the procedure *fire_employee*, as follows:

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
   DELETE FROM emp WHERE empno = emp_id;
END;
```

When creating subprograms, you can use the keyword AS instead of IS in the specification for readability. For more information about creating and using stored subprograms, see *Oracle7 Server Application Developer's Guide*.

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 in some cases, expressions can be used as actual parameters:

```
raise_salary(emp_num, merit + cola);
```

The variables declared in a subprogram specification and referenced in the subprogram body are *formal* parameters. For example, the following procedure declares two formal parameters named *emp_id* and *increase*:

```
PROCEDURE raise_salary (emp_id INTEGER, increase REAL) IS
   current_salary  REAL;
   ...
BEGIN
   SELECT sal INTO current_salary FROM emp WHERE empno = emp_id;
   ...
   UPDATE emp SET sal = sal + increase WHERE empno = emp_id;
END raise_salary;
```

Though not necessary, it is good programming practice 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. Before assigning the value of an actual parameter to a formal parameter, PL/SQL converts the datatype of the value if necessary. For example, the following call to *raise_salary* is legal:

```
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 – 20.

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. For example, given the declarations

```
DECLARE
   acct INTEGER;
   amt  REAL;
   PROCEDURE credit (acctno INTEGER, amount REAL) IS
   BEGIN ... END;
```

you can call the procedure *credit* in four logically equivalent ways:

```
BEGIN
   ...
   credit(acct, amt);                      -- positional notation
   credit(amount => amt, acctno => acct);  -- named notation
   credit(acctno => acct, amount => amt);  -- named notation
   credit(acct, amount => amt);            -- mixed notation
END;
```

Positional Notation

The first procedure call uses positional notation. The PL/SQL compiler associates the first actual parameter, *acct*, with the first formal parameter, *acctno*. And, the compiler associates the second actual parameter, *amt*, with the second formal parameter, *amount*.

Named Notation

The second procedure call uses named notation. The arrow (called an *association operator*) 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. Therefore, 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(acctno => 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. It is poor programming practice to have a function return multiple values. 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 := 10.0;
   service_charge   CONSTANT REAL := 0.50;
BEGIN
   ...
   IF amount < minimum_purchase THEN
      amount := amount + service_charge;  -- causes syntax error
   END IF;
```

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 – 15.

OUT Mode

An OUT parameter lets you return values to the caller of a subprogram. Inside the subprogram, an OUT parameter acts like an uninitialized variable. Therefore, its value cannot be assigned to another variable or reassigned to itself. For instance, the following assignment statement causes a compilation error:

```
PROCEDURE calc_bonus (emp_id IN INTEGER, bonus OUT REAL) IS
   hire_date DATE;
BEGIN
   SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
      WHERE empno = emp_id;
   IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
      bonus := bonus + 500;  -- causes syntax error
   END IF;
```

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 syntax error
```

An OUT actual parameter can have a value before the subprogram is called. However, the value is lost when you call the subprogram. Inside the subprogram, an OUT formal parameter cannot be used in an expression; the only operation allowed on the parameter is to assign it a value.

Before exiting a subprogram, 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.

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. That means you can use an IN OUT formal parameter as if it were a normal 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 IN 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 IN OUT formal parameter must be a variable; it cannot be a constant or an expression. Table 7 – 1 summarizes all you need to know about the 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 an uninitialized variable	formal parameter acts like an initialized variable
formal parameter cannot be assigned a value	formal parameter cannot be used in an expression and 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

Table 7 – 1 Parameter Modes

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 create_dept;
```

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 bind variables in the DEFAULT clause. The following SQL*Plus example raises a *bad bind variable* exception 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 execution, the PL/SQL compiler can choose different methods of parameter passing (copy or reference) for different parameters in the same subprogram call. When the compiler chooses the copy method, the value of an actual parameter is copied into the subprogram. When the compiler chooses the reference method, the address of an actual parameter is passed to the subprogram.

The easy–to–avoid problem of *aliasing* occurs 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. Consider the following example:

```
DECLARE
   rent REAL;
   PROCEDURE raise_rent (increase IN OUT REAL) IS
   BEGIN
      rent := rent + increase;
      /* At this point, if the compiler passed the address
         of the actual parameter to the subprogram, the same
         variable has two names. Thus, the term 'aliasing'. */
      ...
   END raise_rent;
   ...
BEGIN
   ...
   raise_rent(rent);  -- indeterminate
```

Aliasing also occurs when the same actual parameter appears twice in a subprogram call. Unless both formal parameters are IN parameters, the result is indeterminate, as the following example shows:

```
DECLARE
   str VARCHAR2(10);
   PROCEDURE reverse (in_str VARCHAR2, out_str OUT VARCHAR2) IS
   BEGIN
      /* Reverse order of characters in string here. */
      ...
      /* At this point, whether the value of in_str
         is 'abcd' or 'dcba' depends on the methods of
         parameter passing chosen by the compiler. */
   END reverse;
   ...
BEGIN
   str := 'abcd';
   reverse(str, str);  -- indeterminate
```

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. (Figure 2 – 1 on page 2 – 10 shows the datatype families.)

Suppose you want to initialize the first *n* rows in two PL/SQL 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 PL/SQL 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;
```

Also, you might write the next procedure to initialize the PL/SQL 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
   ...
END;
```

Restrictions

Only local or packaged subprograms can be overloaded. Therefore, you cannot overload standalone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode. For example, you cannot overload the following two procedures:

```
PROCEDURE reconcile (acctno IN INTEGER) IS
BEGIN
   ...
END;

PROCEDURE reconcile (acctno 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.

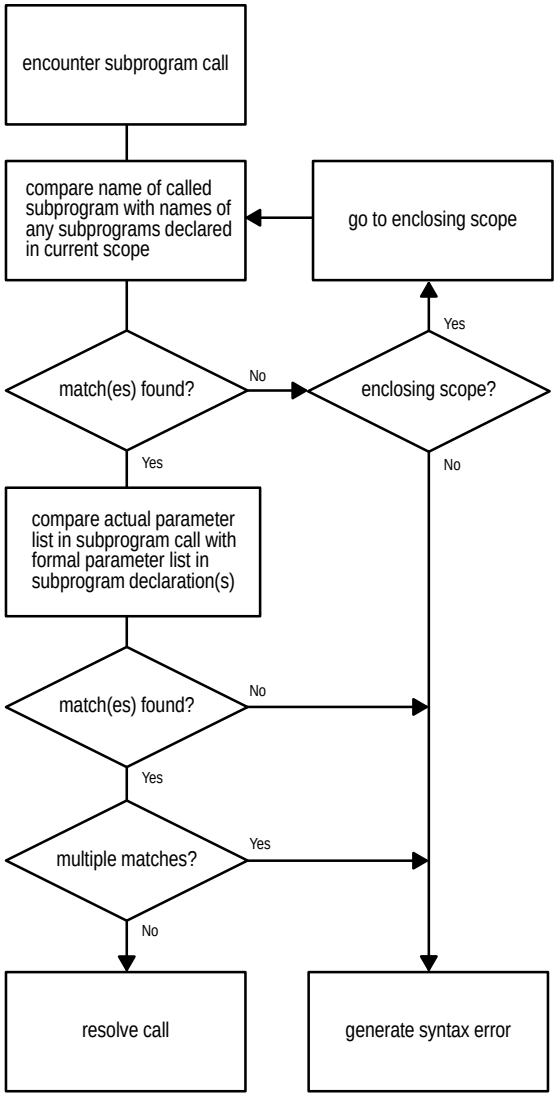

Figure 7 – 1 How the PL/SQL Compiler Resolves Calls

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 swap;
      PROCEDURE swap (n1 REAL, n2 REAL) IS
      BEGIN ... END swap;
   BEGIN
      ...
      swap(date1, date2);
   END valid;
BEGIN
   ...
END swap;
```

Avoiding 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
```

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 objects 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.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. That is, 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.

An Example

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. Recursion is appropriate when the problem can be broken down into simpler versions of itself. For example, you can evaluate 3! as follows:

```
0! = 1
1! = 1 * 0! = 1 * 1 = 1
2! = 2 * 1! = 2 * 1 = 2
3! = 3 * 2! = 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.

Another Example

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. 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.

```
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;
```

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.

Caution

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.

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 – 8.)

```
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 the following 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.

CHAPTER

8 Packages

Good as it is to inherit a library, it is better to collect one.

Augustine Birrell

This chapter shows you how to bundle related PL/SQL programming constructs into a package. The packaged constructs 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, like a library unit, its contents can be shared by many applications.

What Is a Package?

A *package* is a database object that groups logically related PL/SQL types, objects, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The *specification* is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The *body* fully defines cursors and subprograms, and so implements the specification.

Unlike subprograms, packages cannot be called, parameterized, or nested. Still, the format of a package is similar to that of a subprogram:

```
CREATE PACKAGE name AS  -- specification (visible part)
   -- public type and object declarations
   -- subprogram specifications
END [name];

CREATE PACKAGE BODY name AS  -- body (hidden part)
   -- private type and object declarations
   -- subprogram bodies
[BEGIN
   -- initialization statements]
END [name];
```

The specification holds *public* declarations, which are visible to your application. The body holds implementation details and *private* declarations, which are hidden from your application. As Figure 8 – 1 shows, you can think of the specification as an operational interface and of the body as a "black box":

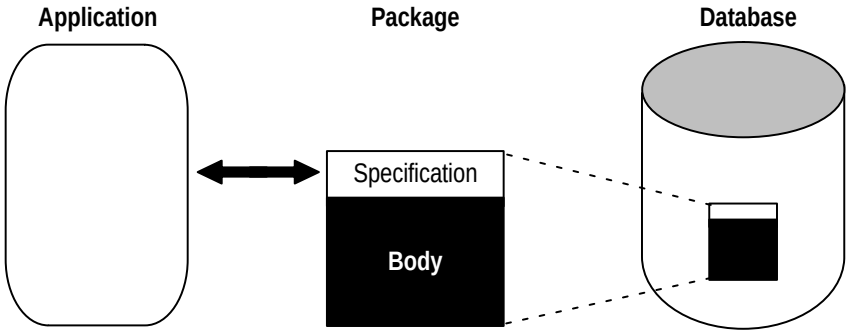

Figure 8 – 1 Package Interface

You can debug, enhance, or replace a package body without changing the interface (package specification) to the package body.

To create packages and store them permanently in an Oracle database, you use the CREATE PACKAGE and CREATE PACKAGE BODY statements, which you can execute interactively from SQL*Plus or Server Manager. For more information, see *Oracle7 Server Application Developer's Guide*.

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 PACKAGE emp_actions AS  -- specification
   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 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 specification 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, objects, 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 specifications. You can code and compile a specification without its body. Once the specification 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.

Information Hiding

With packages, you can specify which types, objects, 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. Therefore, subsequent calls to related subprograms in the package require no disk I/O.

In addition, packages stop cascading dependencies and so avoid unnecessary recompiling. For example, if you change the definition of a standalone function, Oracle must recompile all stored subprograms that call the function. However, 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 Specification

The package specification contains public declarations. The scope of these declarations is local to your database schema and global to the package. So, the declared objects are accessible from your application and from anywhere in the package. Figure 8 – 2 illustrates the scoping.

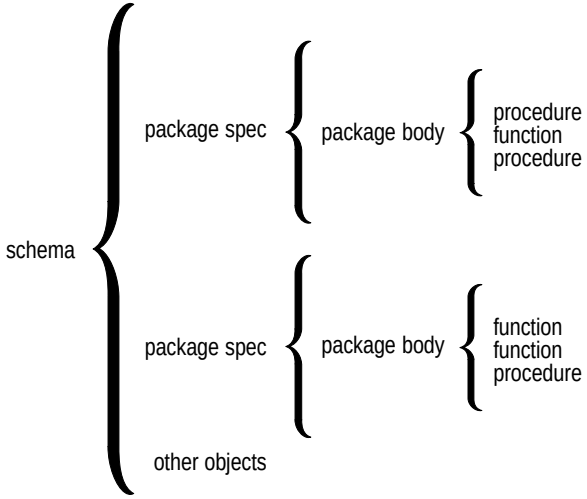

Figure 8 – 2 Package Scope

The specification lists the package resources available to applications. All the information your application needs to use the resources is in the specification. 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 the underlying implementation of *fac* (whether it is iterative or recursive, for example).

Only subprograms and cursors have an underlying implementation or *definition*. So, if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary. Consider the following bodiless package:

```
-- a bodiless package
CREATE PACKAGE trans_data AS
   TYPE TimeTyp IS RECORD (
      minute SMALLINT,
      hour   SMALLINT);
```

```
   TYPE TransTyp IS RECORD (
      category VARCHAR2,
      account  INTEGER,
      amount   REAL,
      time     TimeTyp);
   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, objects, and subprograms declared within a package specification, you use dot notation, as follows:

```
package_name.type_name
package_name.object_name
package_name.subprogram_name
```

You can reference package contents from a database trigger, a stored subprogram, an Oracle Precompiler application, an OCI application, or an Oracle tool such as SQL*Plus. For example, you might call the packaged procedure *hire_employee* from SQL*Plus, as follows:

```
SQL> EXECUTE emp.actions.hire_employee('TATE', 'CLERK', ...);
```

In the following example, you call the same procedure from an anonymous PL/SQL block embedded in a Pro*C program:

```
EXEC SQL EXECUTE
   BEGIN
      emp_actions.hire_employee(:name, :title, ...);
   END;
END-EXEC;
```

The actual parameters *name* and *title* are host variables.

Restriction

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, ...);
   ...
END random;
```

The Package Body

The package body implements the package specification. That is, the package body contains the definition of every cursor and subprogram declared in the package specification. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specifications also appear in the package specification.

To match subprogram specifications 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 calc_bonus;
END emp_actions;
```

The package body can also contain private declarations, which define types and objects necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and objects are inaccessible except from within the package body. Unlike a package specification, 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.

Recall that if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary. However, the body can still be used to initialize objects declared in the specification.

Some Examples

Consider the package below named *emp_actions*. The package specification declares the following types, objects, 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, increase 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, increase 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 + increase
         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 Objects

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 specification 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 specification 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 specification. For example, the constant *minimum_balance* declared in the specification 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

Recall from Chapter 7 that 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.

Package STANDARD

A package named STANDARD defines the PL/SQL environment. The package specification 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 a database trigger, a stored subprogram, an Oracle Precompiler application, an OCI application, and various Oracle tools including Oracle Forms, Oracle Reports, and SQL*Plus.

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

Oracle7 and various Oracle tools are supplied with product–specific packages that help you build PL/SQL–based applications. For example, Oracle7 is supplied with the packages DBMS_STANDARD, DBMS_SQL, DBMS_ALERT, DBMS_OUTPUT, DBMS_PIPE, UTL_FILE, and others. Brief descriptions of these packages follow; for more information, see *Oracle7 Server Application Developer's Guide*.

DBMS_STANDARD Package DBMS_STANDARD provides language facilities that help your application interact with Oracle. For instance, a procedure named *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 – 10.

DBMS_SQL Package DBMS_SQL allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time. For an example, see "Using DDL and Dynamic SQL" on page 5 – 7.

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 *put_line* procedure outputs information to a buffer in the SGA. You display the information by calling the procedure *get_line* or by using the command SET SERVEROUTPUT ON in SQL*Plus or Server Manager. For an example, see "Displaying Output" on page 9 – 6.

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.

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 specifications reflect the design of your application. So, define them before the package bodies. Place in a specification only the types, objects, 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 specification. Changes to a package body do not require Oracle to recompile dependent procedures. However, changes to a package specification require Oracle to recompile every stored subprogram that references the package.

PART

II Language Reference

CHAPTER

9

Execution Environments

Three things are to be looked to in a building: that it stand on the right spot; that it be securely founded; that it be successfully executed.

Goethe

You can use PL/SQL with a variety of application development tools. This chapter shows you how to use PL/SQL in the SQL*Plus, Oracle Precompiler, and OCI environments.

SQL*Plus Environment

After entering the SQL*Plus environment, you can use PL/SQL in several ways:

- input, store, and run a PL/SQL block
- create, load, and run a script containing PL/SQL blocks, subprograms, and/or packages
- pass bind variables to PL/SQL
- call a PL/SQL stored subprogram

Note: This section discusses these topics briefly. For a full discussion, see *SQL*Plus User's Guide and Reference.*

Inputting an Anonymous Block

Every PL/SQL block begins with the keyword DECLARE or, if the block has no declarative part, with the keyword BEGIN. Typing either keyword at the SQL*Plus prompt (SQL>) signals SQL*Plus to

- clear the SQL buffer
- enter INPUT mode
- ignore semicolons (the SQL statement terminator)

SQL*Plus expects you to input an unlabeled PL/SQL block, so you cannot start with a block label.

You input the PL/SQL block line by line. Ending the block with a period (.) on a line by itself stores the block in the SQL buffer.

You can save your PL/SQL block in a script file as follows:

```
SQL> SAVE <filename>
```

If you want to edit the file, you can use the SQL*Plus line editor. For instructions, see *SQL*Plus User's Guide and Reference.* After editing the file, you can save it again as follows:

```
SQL> SAVE <filename> REPLACE
```

Executing an Anonymous Block

After inputting a PL/SQL block, you need not end it with a period. Ending the block with a slash (/) on a line by itself stores the block in the SQL buffer, then runs the block. Once it is stored in the SQL buffer, you can run the PL/SQL block again, as follows:

```
SQL> RUN  or  SQL> /
```

When the block is finished running, you are returned to the SQL*Plus prompt. The SQL buffer is not cleared until you start inputting the next SQL statement or PL/SQL block.

Creating a Script

You can use your favorite text editor to create scripts containing SQL*Plus statements and PL/SQL blocks, subprograms, and/or packages. In the following example, a PL/SQL block is preceded by SQL*Plus statements that set up a report:

```
CLEAR BREAKS
CLEAR COLUMNS
COLUMN ENAME HEADING Name
TTITLE 'CLERICAL STAFF'
DECLARE
   avg_sal NUMBER(7,2);
BEGIN
   SELECT AVG(sal) INTO avg_sal FROM emp;
   IF avg_sal < 1500 THEN
      UPDATE emp SET sal = sal * 1.05 WHERE job = 'CLERK';
   END IF;
END;
/
SELECT ENAME, SAL FROM EMP WHERE JOB = 'CLERK';
```

The two CLEAR statements get rid of any settings left over from a previous report. The COLUMN statement changes the ENAME column heading to Name. The TTITLE statement specifies a title that appears at the top of each page in the report. The semicolon (;) following each SQL*Plus statement executes that statement. Likewise, the slash (/) following the PL/SQL block executes that block.

Loading and Running a Script

After invoking SQL*Plus, you can load and run a script in one step, as follows:

```
SQL> START <filename>  or  SQL> @<filename>
```

Your PL/SQL block can take advantage of the SQL*Plus substitution variable feature. Before running a script, SQL*Plus prompts for the value of any variable prefixed with an ampersand (&). In the following example, SQL*Plus prompts for the value of *num*:

```
SQL> BEGIN
  2     FOR i IN 1..&num LOOP ...
  ...
  8  END;
  9  /
Enter value for num:
```

Creating a Stored Subprogram, Package, or Trigger

To create PL/SQL subprograms, packages, and triggers and store them permanently in an Oracle database, you use the following SQL commands:

- CREATE FUNCTION
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PROCEDURE
- CREATE TRIGGER

When you type any of these commands, SQL*Plus clears the SQL buffer and enters INPUT mode. In the following example, you input a PL/SQL procedure, then create and store it in the database by typing a slash:

```
SQL> CREATE PROCEDURE create_dept (new_name CHAR, new_loc CHAR) AS
  2  BEGIN
  3     INSERT INTO dept
  4        VALUES (deptno_seq.NEXTVAL, new_name, new_loc);
  5  END create_dept;
  6  /
Procedure created.
```

If SQL*Plus tells you that the subprogram, package, or trigger was created with compilation errors, you can view them by typing the SQL*Plus command SHOW ERRORS, as follows:

```
SQL> SHOW ERRORS
```

Using Bind Variables

A *bind variable* is a variable you declare in SQL*Plus, then pass to one or more PL/SQL programs, which can use it like any other variable. Both SQL*Plus and PL/SQL can reference the bind variable, and SQL*Plus can display its value.

To declare a bind variable, you use the SQL*Plus command VARIABLE. In the following example, you declare a variable of type NUMBER:

```
VARIABLE return_code NUMBER
```

Note: If you declare a bind variable with the same name as a PL/SQL program variable, the latter takes precedence.

To reference a bind variable in PL/SQL, you must prefix its name with a colon(:), as the following example shows:

```
:return_code := 0;
IF credit_check_ok(acct_no) THEN
   :return_code := 1;
END IF;
```

To display the value of a bind variable in SQL*Plus, you use the PRINT command, as follows:

```
SQL> PRINT return_code

RETURN_CODE
-----------
          1
```

In the script below, you declare a bind variable of type REFCURSOR. (The SQL*Plus datatype REFCURSOR lets you declare cursor variables, which you can use to return query results from stored subprograms.) 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)
```

Calling Stored Subprograms

From SQL*Plus, you can call standalone and packaged subprograms stored in a local or remote database. For example, you might call the local standalone procedure *create_dept*, as follows:

```
SQL> EXECUTE create_dept('ADVERTISING', 'NEW YORK')
```

This call is equivalent to the following call issued from an anonymous PL/SQL block:

```
SQL> BEGIN create_dept('ADVERTISING', 'NEW YORK'); END;
```

In the next example, you use the database link *newyork* to call the remote stored procedure *raise_salary*:

```
SQL> EXECUTE raise_salary@newyork(7499, 1500)
```

You can create synonyms to provide location transparency for remote standalone procedures.

Displaying Output

Currently, PL/SQL does not support I/O. However, the package DBMS_OUTPUT (supplied with Oracle7) allows you to display output from PL/SQL blocks and subprograms, which makes it easier to test and debug them. The procedure *put_line* lets you output information to a buffer. The SQL*Plus command SET SERVEROUTPUT ON lets you display the information. For example, suppose you create the following stored procedure:

```
CREATE PROCEDURE calc_payroll (payroll IN OUT REAL) 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('payroll: ' || TO_CHAR(payroll));
END calc_payroll;
```

When you issue the following commands, SQL*Plus displays the value of *payroll* calculated by the procedure:

```
SQL> SET SERVEROUTPUT ON
SQL> VARIABLE num NUMBER
SQL> EXECUTE calc_payroll(:num)
```

For more information about package DBMS_OUTPUT, see *Oracle7 Server Application Developer's Guide.*

Oracle Precompiler Environment

The Oracle Precompilers allow you to embed PL/SQL blocks within programs written in any of the following high–level languages: Ada, C, COBOL, FORTRAN, Pascal, and PL/I. Such programs and languages are called *host programs* and *host languages*, respectively.

After writing a program, you precompile the source file. The precompiler checks the program for syntax errors, then generates a modified source file, which can be compiled, linked, and executed in the usual way.

Embedding PL/SQL Blocks

You can embed a PL/SQL block wherever you can embed a SQL statement; the precompiler treats them alike. To embed a PL/SQL block in your host program, you must place the block between the keywords EXEC SQL EXECUTE and END–EXEC, as follows:

```
EXEC SQL EXECUTE
   BEGIN
      ...
   END;
END–EXEC;
```

Be sure to follow the keyword END–EXEC with the host–language statement terminator.

Using Host Variables

You use *host variables* to pass values and status codes back and forth between a host program and an embedded PL/SQL block. You declare host variables in the program *Declare Section* using regular host language syntax. Inside a PL/SQL block, the scope of host variables is global.

Both the host program and the PL/SQL block can set and reference the value of a host variable. The value of an *input* host variable is set by the host program and referenced by Oracle. Conversely, the value of an *output* host variable is set by Oracle and referenced by the host program.

All references to host variables in a PL/SQL block must be prefixed with a colon. That way, the precompiler can tell host variables from PL/SQL variables and database objects.

Some Examples

The Pro*C program below illustrates the use of host variables in a PL/SQL block. The program prompts the user for the name of an employee, then passes the name to an embedded PL/SQL block, which uses the name to query an Oracle database. Finally, the results of the query are passed back to the host program, which displays them.

```
-- available online in file EXAMP9
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR  empname[11];
   VARCHAR  jobtype[9];
   VARCHAR  hired[9];
   int      salary;
   int      dept;
   int      served_longer;
   int      higher_sal;
   int      total_in_dept;
   VARCHAR  uid[20];
   VARCHAR  pwd[20];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

void sqlerror();
main()
{
      /* Set up userid and password */
   strcpy (uid.arr,"scott");
   uid.len = strlen(uid.arr);
   strcpy (pwd.arr,"tiger");
   pwd.len = strlen(pwd.arr);

   printf("\n\n\tEmbedded PL/SQL Demo\n\n");
   printf("Trying to connect...");
      /* Check for SQL errors */
   EXEC SQL WHENEVER SQLERROR DO sqlerror();
      /* Connect to Oracle */
   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
   printf(" connected.\n");

   for (;;) /* Loop indefinitely */
   {
      printf("\n** Name of employee? (<CR> to quit)  ");
      gets(empname.arr);               /* Get the name      */
      if (strlen(empname.arr) == 0)    /* No name entered,  */
      {
         EXEC SQL COMMIT WORK RELEASE; /* so log off Oracle */
         exit(0);                      /* and exit program  */
      }
      empname.len = strlen(empname.arr);
      jobtype.len = 9;
      hired.len = 9;
```

```
      /* ----- Begin PL/SQL block ----- */
      EXEC SQL EXECUTE
      BEGIN
         SELECT job, hiredate, sal, deptno
            INTO :jobtype, :hired, :salary, :dept FROM emp
            WHERE ename = UPPER(:empname);
            /* Get number of people whose length *
             * of service is longer              */
         SELECT COUNT(*) INTO :served_longer FROM emp
            WHERE hiredate < :hired;
            /* Get number of people with a higher salary */
         SELECT COUNT(*) INTO :higher_sal FROM emp
            WHERE sal > :salary;
            /* Get number of people in same department */
         SELECT COUNT(*) INTO :total_in_dept FROM emp
            WHERE deptno = :dept;
      END;
      END-EXEC;
      /* ----- End PL/SQL block ----- */

         /* Null-terminate character strings returned by Oracle */
      jobtype.arr[jobtype.len] = '\0';
      hired.arr[hired.len] = '\0';
         /* Display the information */
      printf("\n%s's job is: %s\n", empname.arr, jobtype.arr);
      printf("Hired on: %s\n", hired.arr);
      printf("    %d people have served longer\n", served_longer);
      printf("Salary is: %d\n", salary);
      printf("    %d people have a higher salary\n", higher_sal);
      printf("Department number is: %d\n", dept);
      printf("    %d people in the department\n", total_in_dept);
   }  /* End of loop */
}

void sqlerror()
{
      /* Avoid infinite loop if rollback causes an error */
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("\nOracle error detected:\n");
      /* Print error message and disconnect from Oracle */
   printf("\n%.70s\n", sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}
```

Notice that the host variable *empname* is assigned a value before the PL/SQL block is entered and that the other host variables are assigned values inside the block. When necessary, Oracle converts between its internal datatypes and standard host–language datatypes.

The next Pro*C example shows how two PL/SQL banking transactions might be implemented:

```
-- available online in file EXAMP10
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
   int     acct, amount;
   VARCHAR tran_type[10];
   VARCHAR status[65];
   VARCHAR uid[20];
   VARCHAR pwd[20];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

void sqlerror();
main()
{
      /* Set up userid and password */
   strcpy (uid.arr,"scott");
   uid.len=strlen(uid.arr);
   strcpy (pwd.arr,"tiger");
   pwd.len=strlen(pwd.arr);

   printf("\n\n\tEmbedded PL/SQL Demo\n\n");
   printf("Trying to connect...");
      /* Check for SQL errors */
   EXEC SQL WHENEVER SQLERROR DO sqlerror();
      /* Connect to Oracle */
   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
   printf(" connected.\n");

   for (;;) /* Loop indefinitely */
   {
      printf("\n\n** Account number? (-1 to quit)");
      scanf("%d", &acct);
      if (acct == -1)   /* Disconnect from Oracle and */
      {                 /* exit program if acct is -1 */
         EXEC SQL COMMIT WORK RELEASE;
         exit(0);
      }
      printf("\n   Transaction type? (C)redit or (D)ebit   ");
      scanf("%s", &tran_type.arr);
      tran_type.len = 1;   /* Only want first character */
```

```
      printf("\n   Transaction amount? (in whole dollars)  ");
      scanf("%d", &amount);

      /* ----- Begin PL/SQL block ----- */
      EXEC SQL EXECUTE
      DECLARE
         old_bal    NUMBER(11,2);
         no_account EXCEPTION;
      BEGIN
         :tran_type := UPPER(:tran_type);
         IF :tran_type = 'C' THEN       -- credit the account
            UPDATE accounts SET bal = bal + :amount
               WHERE account_id = :acct;
            IF SQL%ROWCOUNT = 0 THEN    -- no rows affected
               RAISE no_account;
            ELSE
               :status := 'Credit complete.';
            END IF;
         ELSIF :tran_type = 'D' THEN    -- debit the account
            SELECT bal INTO old_bal FROM accounts
               WHERE account_id = :acct;
            IF old_bal >= :amount THEN  -- has sufficient funds
                UPDATE accounts SET bal = bal - :amount
                   WHERE account_id = :acct;
               :status := 'Debit applied';
            ELSE
               :status := 'Insufficient funds';
            END IF;
         ELSE
            :status := :tran_type || ' is an illegal transaction';
         END IF;
         COMMIT;
      EXCEPTION
         WHEN NO_DATA_FOUND OR no_account THEN
            :status := 'Nonexistent account';
         WHEN OTHERS THEN
            :status := 'Error: ' || SQLERRM(SQLCODE);
      END;
      END-EXEC;
      /* ----- End the PL/SQL block ----- */

      status.arr[status.len] = '\0';  /* null-terminate string */
      printf("\n\n   Status: %s", status.arr);
   }  /* End of loop */
}
```

```
void sqlerror()
{
      /* Avoid infinite loop if rollback causes an error */
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("\nOracle error detected:\n");
      /* Print error message and disconnect from Oracle */
   printf("\n%.70s\n", sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}
```

Using Indicator Variables

You can associate any host variable with an optional indicator variable. An *indicator variable* is an integer variable that indicates the value or condition of a host variable. You use indicator variables to assign nulls to input host variables and to detect nulls or truncated values in output host variables.

For input host variables, the values your program can assign to an indicator variable have the following meanings:

–1	Ignoring the value of the host variable, Oracle will assign a null to the database column.
>= 0	Oracle will assign the value of the host variable to the database column.

For output host variables, the values Oracle can assign to an indicator variable have the following meanings:

–2	Oracle assigned a truncated column value to the host variable but could not store the original length of the column value in the indicator variable because the number was too large.
–1	The database column contains a null, so the value of the host variable is indeterminate.
0	Oracle assigned an intact column value to the host variable.
> 0	Oracle assigned a truncated column value to the host variable and stored the original length of the column value in the indicator variable.

An indicator variable must be defined in the Declare Section as a 2–byte integer and, in SQL statements, must be prefixed with a colon and appended to its host variable unless you use the keyword INDICATOR, as follows:

```
:host_variable INDICATOR :indicator_variable
```

A host language needs indicator variables because it cannot manipulate nulls. PL/SQL meets this need by allowing an embedded PL/SQL block to accept nulls from the host program and return nulls or truncated values to it.

In the following Pro*COBOL example, the PL/SQL block uses an indicator variable to return a null status code to the host program:

```
EXEC SQL EXECUTE
BEGIN
   ...
   SELECT ENAME, COMM INTO :MY-ENAME, :MY-COMM:COMM-IND FROM EMP
      WHERE EMPNO = :MY-EMPNO
END;
END-EXEC.
MOVE MY-COMM TO MY-COMM-OUT.
DISPLAY "Commission: "
   WITH NO ADVANCING.
IF COMM-IND = -1
*  If the value returned by an indicator variable is -1,
*  its output host variable is null.
   DISPLAY "N/A"
ELSE
   DISPLAY MY-COMM-OUT.
```

Inside a PL/SQL block, an indicator variable must be prefixed with a colon and appended to its host variable.

You cannot refer to an indicator variable by itself. Furthermore, if you refer to a host variable with its indicator variable, you must always refer to it that way in the same block. In the next example, because the host variable appears with its indicator variable in the SELECT statement, it must also appear that way in the IF statement:

```
EXEC SQL EXECUTE
DECLARE
   ...
   status_unknown EXCEPTION;
BEGIN
   ...
   SELECT ename, job INTO :my_ename, :my_job:job_ind FROM emp
WHERE empno = :my_empno;
   IF :my_job:job_ind IS NULL THEN
      RAISE status_unknown;
   END IF;
   ...
END;
END-EXEC;
```

Although you cannot refer directly to indicator variables inside a PL/SQL block, PL/SQL checks their values upon entering the block and sets their values correctly upon exiting the block.

Nulls

Upon entering a block, if an indicator variable has a value of –1, PL/SQL assigns a null to the host variable. Upon exiting the block, if a host variable is null, PL/SQL assigns a value of –1 to the indicator variable. In the following example, the exception *name_missing* is raised if the indicator variable *ename_ind* had a value of –1 before the PL/SQL block was entered:

```
EXEC SQL EXECUTE
DECLARE
   ...
   name_missing EXCEPTION;
BEGIN
   ...
   IF :my_ename:ename_ind IS NULL THEN
      RAISE name_missing;
   END IF;
   ...
EXCEPTION
   WHEN name_missing THEN
      ...
END;
END-EXEC;
```

Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string. In the following example, the host program will be able to tell, by checking the value of *ename_ind*, if a truncated value was assigned to *my_ename*:

```
EXEC SQL EXECUTE
DECLARE
   new_ename CHAR(10);
   ...
BEGIN
   ...
   :my_ename:ename_ind := new_ename;
   ...
END;
END-EXEC;
```

Using the VARCHAR Pseudotype

You can use the VARCHAR pseudotype to declare variable–length character strings. (A *pseudotype* is a datatype not native to your host language.) VARCHAR variables have a 2–byte length field followed by a string field of up to 65533 bytes. For example, the Pro*C Precompiler expands the declaration

```
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR my_ename[10]
EXEC SQL END DECLARE SECTION;
```

into the following data structure:

```
struct {
   unsigned short len;
   unsigned char  arr[10];
}  my_ename;
```

To get the length of a VARCHAR, simply refer to its length field. You need not use a string function or character–counting algorithm.

Oracle automatically sets the length field of a VARCHAR output host variable. However, to use a VARCHAR output host variable in your PL/SQL block, you must initialize the length field *before* entering the PL/SQL block. So, set the length field to the declared (maximum) length of the VARCHAR, as shown in the following Pro*C example:

```
EXEC SQL BEGIN DECLARE SECTION;
   int     my_empno;
   VARCHAR my_ename[10] /* declare variable-length string */
   float   my_sal;
   ...
EXEC SQL END DECLARE SECTION;
   ...
   my_ename.len = 10;   /* initialize length field */
   EXEC SQL EXECUTE
   BEGIN
      SELECT ename, sal INTO :my_ename, :my_sal FROM emp
         WHERE empno = :my_empno;
      ...
   END;
   END-EXEC;
```

Using the DECLARE TABLE Statement

If an embedded PL/SQL block refers to a database table that does not yet exist, the precompiler generates an error. To avoid such errors, you can use the DECLARE TABLE statement to tell the precompiler what the table will look like. In the following Pro*C example, you declare the *dept* table:

```
EXEC SQL DECLARE dept TABLE(
   deptno NUMBER(2),
   dname  VARCHAR2(14),
   loc    VARCHAR2(13));
```

If you use DECLARE TABLE to define a table that already exists, the precompiler uses your definition, ignoring the one in the Oracle data dictionary. Note that you cannot use the DECLARE TABLE statement inside a PL/SQL block.

Using the SQLCHECK Option

The Oracle Precompilers can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. You control the level of checking by entering the SQLCHECK option inline or on the command line. You can specify the following values for SQLCHECK:

- SEMANTICS
- SYNTAX
- NONE

However, if you embed PL/SQL blocks in your program, you must specify SQLCHECK=SEMANTICS. When SQLCHECK=SEMANTICS, the precompiler checks the syntax and semantics of SQL data manipulation statements and PL/SQL blocks.

The precompiler gets information needed for the semantic check by using embedded DECLARE TABLE statements or by connecting to Oracle and accessing the data dictionary. So, unless every database table referenced in a SQL statement or PL/SQL block is defined by a DECLARE TABLE statement., you must specify the option USERID on the command line. For more information see *Programmer's Guide to the Oracle Precompilers*.

Using Dynamic SQL

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to (or built by) the program at run time. For example, they might be entered interactively or read from a file.

The Oracle Precompilers treat a PL/SQL block like a single SQL statement. So, like a SQL statement, a PL/SQL block can be stored in a string host variable for processing by dynamic SQL commands.

However, recall from Chapter 2 that you cannot use single–line comments in a PL/SQL block that will be processed dynamically. Instead, use multi–line comments.

Following is a brief look at how PL/SQL is used with dynamic SQL Methods 1, 2, and 4. For more information, see *Programmer's Guide to the Oracle Precompilers*.

With Method 1

If your PL/SQL block contains no host variables, you can use Method 1 to execute the PL/SQL string in the usual way. In the following Pro*C example, you prompt the user for a PL/SQL block, store it in a string host variable named *user_block*, then execute it:

```
main()
{
   printf("\nEnter a PL/SQL block: ");
   scanf("%s", user_block);
   EXEC SQL EXECUTE IMMEDIATE :user_block;
```

When you store a PL/SQL block in a string host variable, omit the keywords EXEC SQL EXECUTE, the keyword END–EXEC, and the statement terminator.

With Method 2

If your PL/SQL block contains a known number of input and output host variables, you can use dynamic SQL Method 2 to prepare and execute the PL/SQL string in the usual way. In the Pro*C example below, the PL/SQL block uses one host variable named *my_empno*. The program prompts the user for a PL/SQL block, stores it in a string host variable named *user_block*, then prepares and executes the block:

```
main()
{
   printf("\nEnter a PL/SQL block: ");
   scanf("%s", user_block);
   EXEC SQL PREPARE my_block FROM :user_block;
   EXEC SQL EXECUTE my_block USING :my_empno;
```

Note that *my_block* is an identifier used by the precompiler, *not* a host or program variable.

The precompiler treats all PL/SQL host variables as *input* host variables whether they serve as input or output host variables (or both) inside the PL/SQL block. So, you must put *all* host variables in the USING clause.

When the PL/SQL string is executed, host variables in the USING clause replace corresponding placeholders in the prepared string. Although the precompiler treats all PL/SQL host variables as input host variables, values are assigned correctly. Input (program) values are assigned to input host variables, and output (column) values are assigned to output host variables.

With Method 4

If your PL/SQL block contains an unknown number of input or output host variables, you must use Method 4. To do so, you set up a *bind descriptor* for all the input and output host variables. Executing the DESCRIBE BIND VARIABLES statement stores information about input *and* output host variables in the bind descriptor.

Mimicking Dynamic SQL

Without dynamic SQL, you cannot use PL/SQL variables in a query to specify database columns. Consider the SELECT statement below, which does *not* assign a value from the *ename* database column to the variable *my_ename*. Instead, if the *emp* table has a column named *colx*, a value from that column is assigned to *my_ename*. If the table has no such column, the value of PL/SQL variable *colx* (that is, the string value 'ename') is assigned to *my_ename*.

```
DECLARE
   colx     VARCHAR2(10);
   my_ename VARCHAR2(10);
   ...
BEGIN
   colx := 'ename';
   SELECT colx INTO my_ename FROM emp WHERE ...
   ...
END;
```

However, you can mimic dynamic SQL by using the DECODE function. In the following example, the data returned depends on the value of *my_column*:

```
DECLARE
   my_column VARCHAR2(10);
   my_data   emp.ename%TYPE;
BEGIN
   ...
   my_column := 'hiredate';
   ...
   SELECT DECODE(my_column, 'ename', ename, 'hiredate',
         TO_CHAR(hiredate, 'ddmmyy'), 'empno', empno)
      INTO my_data FROM emp WHERE ... ;
END;
```

The value that DECODE returns is always forced to the datatype of the first result expression. In this example, the first result expression is *ename*, which has datatype VARCHAR2, so the returned value is forced to type VARCHAR2. Thus, *my_data* is correctly declared as *emp.ename*%TYPE.

You can use this technique in many environments. For example, it works in SQL*Plus and Oracle Forms.

Calling Stored Subprograms

To call a stored subprogram from a host program, you must use an anonymous PL/SQL block. In the following example, you call the standalone procedure *create_dept*:

```
EXEC SQL EXECUTE
BEGIN
    create_dept(:number, :name, :location);
END;
END-EXEC;
```

Notice that the actual parameters *number*, *name*, and *location* are host variables.

In the next example, the procedure *create_dept* is part of a package named *emp_actions*, so you must use dot notation to qualify the procedure call:

```
EXEC SQL EXECUTE
BEGIN
    emp_actions.create_dept(:number, :name, :location);
END;
END-EXEC;
```

OCI Environment

The OCI processes SQL statements and PL/SQL blocks similarly with one exception. Inside a PL/SQL block, you must use the OBNDRA, OBINDPS, or OBNDRV call, not ODEFIN or ODEFINPS, to bind all placeholders in a SQL or PL/SQL statement. This holds for both input and output placeholders. The ODEFIN and ODEFINPS calls are *not* supported for PL/SQL blocks.

In PL/SQL, all queries must have an INTO clause containing placeholders (host variables and/or PL/SQL variables) that correspond to items in the select list. For example, the following SELECT statement is not valid inside a PL/SQL block:

```
SELECT ename, sal FROM emp;
```

Instead, it must be coded as follows:

```
SELECT ename, sal INTO :my_ename, :my_sal FROM emp;
```

In the last statement, *my_ename* and *my_sal* are SQL placeholders that correspond to the *ename* and *sal* columns in the select list. You must bind these placeholders using the OBNDRA, OBINDPS, or OBNDRV call. You can bind host arrays to PL/SQL tables using the OBNDRA or OBINDPS call.

Also, you must use named placeholders such as *my_ename* in PL/SQL blocks. Numbered placeholders such as *10* and the corresponding OBNDRN call are *not* supported for PL/SQL blocks.

A Complete Example

The OCI program below, which is written in C, shows how two PL/SQL banking transactions might be implemented. You can find listings of the header files **ocidfn.h** and **ocidem.h**, in *Programmer's Guide to the Oracle Call Interface*.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <oratypes.h>
#include <ocidfn.h>
#ifdef __STDC__
#include <ociapr.h>
#else
#include <ocikpr.h>
#endif
#include <ocidem.h>
Cda_Def cda;
Lda_Def lda;
ub1  hda[256];
text  sqlstm[2048];
void error_handler();

main()
{
   int    acct_number;
   text   trans_type[1];
   float  trans_amt;
   text   status[80];
   if (olog(&lda, hda, "scott/tiger", -1, (text *) 0, -1,
      (text *) 0, -1, OCI_LM_DEF))
   {
      printf("Connect failed.\n");
      exit(EXIT_FAILURE);
   }
   if (oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1))
   {
      printf("Error opening cursor.  Exiting...\n");
      exit(EXIT_FAILURE);
   }
   printf("\nConnected to Oracle.\n");
   /* Construct a PL/SQL block. */
   strcpy(sqlstm, "DECLARE\
         old_bal      NUMBER(9,2);\
         err_msg      CHAR(70);\
         nonexistent  EXCEPTION;\
```

```
   BEGIN\
      :xtrans_type := UPPER(:xtrans_type);\
      IF :xtrans_type = 'C' THEN\
         UPDATE ACCTS SET BAL = BAL + :xtrans_amt\
            WHERE ACCTID = :xacct_number;\
         IF SQL%ROWCOUNT = 0 THEN\
            RAISE nonexistent;\
         ELSE\
            :xstatus := 'Credit applied';\
         END IF;\
      ELSIF :xtrans_type = 'D' THEN\
         SELECT BAL INTO old_bal FROM accts\
            WHERE acctid = :xacct_number;\
         IF old_bal = :xtrans_amt THEN\
            UPDATE accts SET bal = bal - :xtrans_amt\
               WHERE acctid = :xacct_number;\
            :xstatus := 'Debit applied';\
         ELSE\
            :xstatus := 'Insufficient funds';\
         END IF;\
      ELSE\
         :xstatus := 'Invalid type: ' || :xtrans_type;\
      END IF;\
      COMMIT;\
   EXCEPTION\
      WHEN NO_DATA_FOUND OR nonexistent THEN\
         :xstatus := 'Nonexistent account';\
      WHEN OTHERS THEN\
         err_msg := SUBSTR(SQLERRM, 1, 70);\
         :xstatus := 'Error: ' || err_msg;\
   END;");

/* Parse the PL/SQL block. */
if (oparse(&cda, sqlstm, -1, 0, 2))
{
   error_handler(&cda);
   exit(EXIT_FAILURE);
}

/* Bind the status variable. */
if (obndrv(&cda,
   ":xstatus",
   -1,
   status,
   70,
   5,
   -1,
   (text *) 0,
   (text *) 0, -1, -1))
```

```
{
   error_handler(&cda);
   exit(EXIT_FAILURE);
}

/* Bind the transaction type variable. */
if (obndrv(&cda,
   ":xtrans_type",
   -1,
   trans_type,
   1,
   1,
   -1,
   (text *) 0,
   (text *) 0, -1, -1))
{
    error_handler(&cda);
    exit(EXIT_FAILURE);
}

/* Bind the account number. */
if (obndrv(&cda,
   ":xacct_number",
   -1,
   &acct_number,
   sizeof (int),
   3,
   -1,
   (text *) 0,
   (text *) 0, -1, -1))
{
   error_handler(&cda);
   exit(EXIT_FAILURE);
}

/* Bind the transaction amount variable. */
if (obndrv(&cda,
   ":xtrans_amt",
   -1,
   &trans_amt,
   sizeof (float),
   4,
   -1,
   (text *) 0,
   (text *) 0, -1, -1))
{
   error_handler(&cda);
   exit(EXIT_FAILURE);
}
```

```
   for (;;)
   {
      printf("\nAccount number: ");
      scanf("%d", &acct_number);
      fflush(stdin);
      if (acct_number == 0)
         break;
      printf("Transaction type (D or C): ");
      scanf("%c", trans_type);
      fflush(stdin);
      printf("Transaction amount:        ");
      scanf("%f", &trans_amt);
      fflush(stdin);
      /* Execute the block. */
      if (oexec(&cda))
         error_handler(&cda);
      printf("%s\n", status);
   }
   printf("Have a good day!\n");
   exit(EXIT_SUCCESS);
}

void
error_handler(cursor)
    Cda_Def *cursor;
{
    sword n;
    text msg[512];
    printf("\n-- ORACLE error--\n");
    printf("\n");
    n = oerhms(&lda, cursor->rc, msg, (sword) sizeof msg);
    fprintf(stderr, "%s\n", msg);
    if (cursor->fc > 0)
        fprintf(stderr, "Processing OCI function %s",
            oci_func_tab[cursor->fc]);
}
```

Calling Stored Subprograms

To call a stored subprogram from an OCI program, you must use an anonymous PL/SQL block. In the following C example, a call to the standalone procedure *raise_salary* is copied into the string variable *plsql_block*:

```
strcpy(plsql_block, "BEGIN raise_salary(:emp_id, :amount); END;");
```

Then, the PL/SQL string can be bound and executed like a SQL statement.

CHAPTER

10 Language Elements

Grammar, which knows how to control even kings.

Molière

This chapter is a quick reference guide to PL/SQL syntax; 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.

The following sections are arranged alphabetically for easy reference:

Assignment Statement
Blocks
CLOSE Statement
Comments
COMMIT Statement
Constants and Variables
Cursor Attributes
Cursors
Cursor Variables
DELETE Statement
EXCEPTION_INIT Pragma
Exceptions
EXIT Statement
Expressions
FETCH Statement
Functions
GOTO Statement
IF Statement
INSERT Statement
Literals
LOCK TABLE Statement
LOOP Statement
NULL Statement
OPEN Statement
OPEN–FOR Statement
Packages
PL/SQL Table Attributes
PL/SQL Tables
Procedures
RAISE Statement
Records
RETURN Statement
ROLLBACK Statement
%ROWTYPE Attribute
SAVEPOINT Statement
SELECT INTO Statement
SET TRANSACTION Statement
SQL Cursor
SQLCODE Function
SQLERRM Function
%TYPE Attribute
UPDATE Statement

Each of these sections has some or all of the following subsections:

Description
Syntax
Keyword and Parameter Description
Usage Notes
Examples
Related Topics

The syntax of PL/SQL is described using a simple variant of Backus–Naur Form (BNF). BNF is a metalanguage used mainly to define the syntax of computer languages. If you are unfamiliar with BNF, do not worry. The next section tells you all you need to know.

Reading Syntax Definitions

This chapter is *not* meant to provide a formal language definition. So, it defines syntax using a BNF–style grammar less concise but more readable than BNF.

When you are unsure of the syntax to use in a PL/SQL statement, trace through its syntax definition, reading from left to right and top to bottom. You can verify or construct any PL/SQL statement that way.

Syntax definitions use the following symbols and lexical conventions:

::=	This symbol means “is defined as.”
[ ]	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.
lower case	Lower case denotes a syntactic element for which you must substitute a literal, identifier, or construct, whichever is appropriate.
UPPER CASE	Upper case denotes PL/SQL keywords, which must be spelled as shown but can be entered in lower or mixed case.
punctuation	Punctuation other than brackets, braces, vertical bars, and ellipses must be entered as shown.

Assignment Statement

Description

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 – 32.

Syntax

assignment_statement ::=

```
{  cursor_variable_name
 | :host_cursor_variable_name
 | :host_variable_name[:indicator_name]
 | parameter_name
 | plsql_table_name(index)
 | record_name.field_name
 | variable_name} := expression;
```

Keyword and Parameter Description

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. For more information, see "Using Host Variables" on page 9 – 7.

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. For more information, see "Using Indicator Variables" on page 9 – 12.

parameter_name — This identifies a formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears.

plsql_table_name	This identifies a PL/SQL table previously declared within the current scope.
index	This is a numeric expression that must yield a value of type BINARY_INTEGER or a value implicitly convertible to that datatype. For more information, see “Datatype Conversion” on page 2 – 20.
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 10 – 41. 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 and FALSE and the non–value 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 PL/SQL table:

```
plsql_table_name(index) := expression;
```

In the following example, you assign the uppercase value of *last_name* to the third row in PL/SQL 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

Description

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 PL/SQL 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 – 3 and “Scope and Visibility” on page 2 – 30.

Syntax

plsql_block ::=

```
[<<label_name>>]
[DECLARE
   object_declaration [object_declaration] ...
   [subprogram_declaration [subprogram_declaration] ...]]
BEGIN
   seq_of_statements
[EXCEPTION
   exception_handler [exception_handler] ...]
END [label_name];
```

object_declaration ::=

```
{  constant_declaration
 | cursor_declaration
 | cursor_variable_declaration
 | exception_declaration
 | plsql_table_declaration
 | record_declaration
 | variable_declaration}
```

subprogram_declaration ::=

```
{function_declaration | procedure_declaration}
```

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* 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. Objects 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 object 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 objects.

constant_declaration

This construct declares a constant. For the syntax of *constant_declaration*, see "Constants and Variables" on page 10 – 16.

cursor_declaration

This construct declares an explicit cursor. For the syntax of *cursor_declaration*, see "Cursors" on page 10 – 23.

cursor_variable_declaration

This construct declares a cursor variable. For the syntax of *cursor_variable_declaration*, see "Cursor Variables" on page 10 – 27.

exception_declaration

This construct declares an exception. For the syntax of *exception_declaration*, see "Exceptions" on page 10 – 36.

plsql_table_declaration

This construct declares a PL/SQL table. For the syntax of *plsql_table_declaration*, see "PL/SQL Tables" on page 10 – 82.

record_declaration

This construct declares a user–defined record. For the syntax of *record_declaration*, see “Records” on page 10 – 93.

variable_declaration

This construct declares a variable. For the syntax of *variable_declaration*, see “Constants and Variables” on page 10 – 16.

function_declaration

This construct declares a function. For the syntax of *function_declaration*, see “Functions” on page 10 – 51.

procedure_declaration

This construct declares a procedure. For the syntax of *procedure_declaration*, see “Procedures” on page 10 – 87.

BEGIN

This keyword signals the start of the executable part of a PL / SQL block, which contains executable statements. The executable part of a PL / SQL block is required. That is, a block must contain at least one executable statement. The NULL statement meets this requirement.

seq_of_statements

This represents a sequence of executable (not declarative) statements, which can include SQL statements and PL / SQL blocks (sometimes called block statements). The syntax of *seq_of_statements* follows:

```
seq_of_statements ::=

statement [statement] ...
```

Statements are used to create algorithms. Besides SQL statements, PL / SQL has flow–of–control and error–handling statements. PL / SQL statements are free format. That is, they can continue from line to line, providing you do not split keywords, delimiters, or literals across lines. A semicolon (;) must terminate every PL / SQL statement. The syntax of *statement* follows:

```
statement ::=

[<<label_name>>]
{  assignment_statement
 | exit_statement
 | goto_statement
 | if_statement
 | loop_statement
 | null_statement
 | plsql_block
 | raise_statement
 | return_statement
 | sql_statement}
```

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. The syntax of *sql_statement* follows:

```
sql_statement ::=

{  close_statement
 | commit_statement
 | delete_statement
 | fetch_statement
 | insert_statement
 | lock_table_statement
 | open_statement
 | open–for_statement
 | rollback_statement
 | savepoint_statement
 | select_statement
 | set_transaction_statement
 | update_statement}
```

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 10 – 36.

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

Description

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 – 9.

Syntax

close_statement ::=

```
CLOSE {  cursor_name
       | cursor_variable_name
       | :host_cursor_variable_name};
```

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

Comments

Description

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 – 8.

Syntax

comment ::=

```
{-- text | /* text */}
```

Usage Notes

Comments can appear within a statement at the end of a line. However, you cannot nest comments.

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

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

```
-- UPDATE dept SET loc = my_loc WHERE deptno = my_deptno;
```

You can use multi–line comment delimiters to comment–out whole sections of code.

Examples

The following examples show various comment styles:

```
-- compute the area of a circle
area := pi * radius**2;  -- pi equals 3.14159

/* Compute the area of a circle. */
area := pi * radius**2;  /* pi equals 3.14159 */

/*
  The following line computes the area of a circle using pi,
  which is the ratio between the circumference and diameter.
  Pi is an irrational number, meaning that it cannot be
  expressed as the ratio between two integers.
*/
area := pi * radius**2;
```

COMMIT Statement

Description

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 – 39.

Syntax

```
commit_statement ::=

COMMIT [WORK] [COMMENT 'text'];
```

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 cursor that was declared using FOR UPDATE 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 – 45.

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 *Oracle7 Server Concepts*.

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
```

With embedded SQL, the optional RELEASE parameter is allowed after COMMIT WORK. The keyword RELEASE acts like a "disconnect" statement, which logs you off the database once your transaction is committed. PL/SQL does not support data control statements such as CONNECT, GRANT, or REVOKE. Therefore, it does not support the RELEASE parameter.

Related Topics ROLLBACK Statement, SAVEPOINT Statement

Constants and Variables

Description

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 – 22.

Syntax

constant_declaration ::=

```
constant_name CONSTANT
   {  record_name.field_name%TYPE
    | scalar_type_name
    | table_name.column_name%TYPE
    | variable_name%TYPE} [NOT NULL] {:= | DEFAULT} expression;
```

variable_declaration ::=

```
variable_name
   {  cursor_name%ROWTYPE
    | cursor_variable_name%TYPE
    | plsql_table_name%TYPE
    | record_name%TYPE
    | record_name.field_name%TYPE
    | scalar_type_name
    | table_name%ROWTYPE
    | table_name.column_name%TYPE
    | variable_name%TYPE} [[NOT NULL] {:= | DEFAULT} expression];
```

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.

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. For naming conventions, see "Identifiers" on page 2 – 4.
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.
plsql_table_name	This identifies a PL / SQL table previously declared within the current scope.
record_name	This identifies a user–defined record previously declared within the current scope.
table_name	This identifies a database table (or view) that must be accessible when the declaration is elaborated.
%ROWTYPE	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 field, record, PL / SQL table, 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. For the syntax of *expression*, see "Expressions" on page 10 – 41. 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 specification 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

Description

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 – 33.

There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including single–row queries. For multi–row queries, you can explicitly declare a cursor or cursor variable to process the rows.

Syntax

```
cursor_attribute ::=

{  cursor_name
 | cursor_variable_name
 | :host_cursor_variable_name
 | SQL}{%FOUND | %ISOPEN | %NOTFOUND | %ROWCOUNT}
```

Keyword and Parameter Description

cursor_name — This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name — This identifies a PL/SQL cursor variable (or parameter) previously declared within the current scope.

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 10 – 108.

%FOUND — This is a cursor attribute, which can be appended to the name of a cursor or cursor variable. After a cursor is opened but before the first fetch, *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. After a cursor is opened but before the first fetch, *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

You can use the cursor attributes in procedural statements but *not* in SQL statements.

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.

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

Cursors

Description

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 – 9.

Syntax

cursor_declaration ::=

```
CURSOR cursor_name [(cursor_parameter_declaration[,
  cursor_parameter_declaration]...)] IS select_statement;
```

cursor_specification ::=

```
CURSOR cursor_name [(cursor_parameter_declaration[,
  cursor_parameter_declaration]...)]
    RETURN {  cursor_name%ROWTYPE
            | record_name%TYPE
            | record_type_name
            | table_name%ROWTYPE};
```

cursor_body ::=

```
CURSOR cursor_name [(cursor_parameter_declaration[,
  cursor_parameter_declaration]...)]
    RETURN {  cursor_name%ROWTYPE
            | record_name%TYPE
            | record_type_name
            | table_name%ROWTYPE} IS select_statement;
```

cursor_parameter_declaration ::=

```
cursor_parameter_name [IN]
   {  cursor_name%ROWTYPE
    | cursor_variable_name%TYPE
    | plsql_table_name%TYPE
    | record_name%TYPE
    | scalar_type_name
    | table_name%ROWTYPE
    | table_name.column_name%TYPE
    | variable_name%TYPE} [{:= | DEFAULT} expression]
```

Keyword and Parameter Description

cursor_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.
select_statement	This is a query that returns a result set of rows. If the cursor declaration declares parameters, each parameter must be used in the query. The syntax of *select_statement* is like that of *select_into_statement*, which is defined in "SELECT INTO Statement" on page 10 – 104, except that *select_statement* cannot have an INTO clause.
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 specification. Also, the number, order, and datatypes of select items in the SELECT clause must match the RETURN clause.
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. For more information, see "User–Defined Records" on page 4 – 19.
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 field, record, PL/SQL table, database column, or variable.
cursor_variable_name	This identifies a PL/SQL cursor variable previously declared within the current scope.
scalar_type_name	This identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2, which must be specified without constraints. For more information, see "Datatypes" on page 2 – 10.
plsql_table_name	This identifies a PL/SQL table 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 10 – 41. 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 – 30.

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

Two examples of cursor declarations follow:

```
CURSOR c1 IS
   SELECT ename, job, sal FROM emp WHERE deptno = 20;
CURSOR c2 (start_date DATE) IS
   SELECT empno, sal FROM emp WHERE hiredate > start_date;
```

Related Topics

CLOSE Statement, FETCH Statement, OPEN Statement, SELECT INTO Statement

Cursor Variables

Description

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. Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some object instead of the object itself. So, declaring a cursor variable creates a pointer, *not* an object. For more information, see "Using Cursor Variables" on page 5 – 17.

To create cursor variables, you take two steps. First, you define a REF CURSOR type, then declare cursor variables of that type.

Syntax

ref_type_definition ::=

```
TYPE ref_type_name IS REF CURSOR
   RETURN {  cursor_name%ROWTYPE
           | cursor_variable_name%ROWTYPE
           | record_name%TYPE
           | record_type_name
           | table_name%ROWTYPE};
```

cursor_variable_declaration ::=

```
cursor_variable_name ref_type_name;
```

Keyword and Parameter Description

ref_type_name — This is a user–defined type specifier, which is used in subsequent declarations of PL/SQL cursor variables. For naming conventions, see "Identifiers" on page 2 – 4.

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. Currently, cursor variables are the only REF variables that you can declare.

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 previously declared cursor or 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. For more information, see "User–Defined Records" on page 4 – 19.
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 or 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 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. They 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.

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 – 33.

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 the following restrictions, some of which future releases of PL/SQL will remove:

- You cannot declare cursor variables in a package because they do not have persistent state.
- 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 (bind 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.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected.
- You cannot use a REF CURSOR type to specify the element type of a PL/SQL table, which means that elements in a PL/SQL table cannot store the values of cursor variables.
- You cannot use cursor variables with dynamic SQL.

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 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 standalone procedure to open the cursor variable. Simply define the REF CURSOR type in a separate package, then reference that type in the standalone procedure. For instance, if you create the following (bodiless) package, you can create standalone procedures that reference the types it defines:

```
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

DELETE Statement

Description

The DELETE statement removes entire rows of data from a specified table or view. For a full description of the DELETE statement, see *Oracle7 Server SQL Reference.*

Syntax

```
delete_statement ::=

DELETE [FROM] {table_reference | (subquery)} [alias]
   [WHERE {search_condition | CURRENT OF cursor_name}];

table_reference ::=

[schema_name.]{table_name | view_name}[@dblink_name]
```

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 value or set of values to the DELETE statement. The syntax of *subquery* is like the syntax of *select_into_statement* defined in "SELECT INTO Statement" on page 10 – 104, except that *subquery* cannot have an INTO clause.

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.

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

Example

The following statement deletes from the *bonus* table all employees whose sales were below quota:

```
DELETE FROM bonus WHERE sales_amt < quota;
```

Related Topics

FETCH Statement, SELECT Statement

EXCEPTION_INIT Pragma

Description

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 – 9.

Syntax

```
exception_init_pragma ::=

PRAGMA EXCEPTION_INIT (exception_name, error_number);
```

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 *insufficient_privileges* with Oracle error –1031:

```
DECLARE
   insufficient_privileges EXCEPTION;
   PRAGMA EXCEPTION_INIT(insufficient_privileges, –1031);
BEGIN
   ...
EXCEPTION
   WHEN insufficient_privileges THEN
      -- handle the error
END;
```

Related Topics

Exceptions, SQLCODE Function

Exceptions

Description

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.

Syntax

exception_declaration ::=

```
exception_name EXCEPTION;
```

exception_handler ::=

```
WHEN {exception_name [OR exception_name] ... | OTHERS}
   THEN seq_of_statements
```

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.

Each WHEN clause can associate a different sequence of statements with a list of exceptions. However, an exception name can appear only once in the exception–handling part of a PL/SQL block or subprogram.

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.

seq_of_statements

This construct represents a sequence of statements. For the syntax of *seq_of_statements*, see “Blocks” on page 10 – 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 – 5. 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 impossible or undesirable 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 – 10.

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

EXIT Statement

Description

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" on page 3 – 6.

Syntax

```
exit_statement ::=

EXIT [label_name] [WHEN boolean_expression];
```

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 10 – 41.

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 in the normal way:

```
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;
   ...
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

Description

An expression is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of 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 – 33.

Syntax

expression ::=

```
[(]{  boolean_expression
    | character_expression
    | date_expression
    | numeric_expression}[)]
```

boolean_expression ::=

```
[NOT] {  boolean_constant_name
       | boolean_function_call
       | boolean_literal
       | boolean_variable_name
       | other_boolean_form}
[{AND | OR} [NOT] {  boolean_constant_name
                   | boolean_function_call
                   | boolean_literal
                   | boolean_variable_name
                   | other_boolean_form}] ...
```

other_boolean_form ::=

```
expression
   {  relational_operator expression
    | IS [NOT] NULL
    | [NOT] LIKE pattern
    | [NOT] BETWEEN expression AND expression
    | [NOT] IN (expression[, expression]...)
    | {  cursor_name
       | cursor_variable_name
       | :host_cursor_variable_name
       | SQL}{%FOUND | %ISOPEN | %NOTFOUND}
    | plsql_table_name.EXISTS(index)}
```

numeric_expression ::=

```
{  {  cursor_name
    | cursor_variable_name
    | :host_cursor_variable_name
    | SQL}%ROWCOUNT
 | :host_variable_name[:indicator_name]
 | numeric_constant_name
 | numeric_function_call
 | numeric_literal
 | numeric_variable_name
 | plsql_table_name{  .COUNT
                    | .FIRST
                    | .LAST
                    | .NEXT(index)
                    | .PRIOR(index)}}[**exponent]
[ {+ | – | * | /}
 {  {  cursor_name
     | cursor_variable_name
     | :host_cursor_variable_name
     | SQL}%ROWCOUNT
  | :host_variable_name[:indicator_name]
  | numeric_constant_name
  | numeric_function_call
  | numeric_literal
  | numeric_variable_name
  | plsql_table_name{  .COUNT
                     | .FIRST
                     | .LAST
                     | .NEXT(index)
                     | .PRIOR(index)}}[**exponent]]...
```

character_expression ::=

```
{  character_constant_name
 | character_function_call
 | character_literal
 | character_variable_name
 | :host_variable_name[:indicator_name]}
[ || {  character_constant_name
     | character_function_call
     | character_literal
     | character_variable_name
     | :host_variable_name[:indicator_name]}]...
```

```
date_expression ::=

{  date_constant_name
 | date_function_call
 | date_literal
 | date_variable_name
 | :host_variable_name[:indicator_name]}
[{+ | –} 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 the truth tables on page 2 – 34. 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 – 34.
boolean_constant_name	This identifies a constant of type BOOLEAN, which must be initialized to the value TRUE or FALSE or the non–value 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 or FALSE or the non–value 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 and FALSE and the non–value 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 – 36.

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 – 33.
plsql_table_name	This identifies a PL/SQL table previously declared within the current scope.

EXISTS, COUNT, FIRST, LAST, NEXT, PRIOR	These are PL/SQL table attributes. When appended to the name of a PL/SQL table, these attributes return useful information. For example, EXISTS(*n*) returns TRUE if the *n*th element of a PL/SQL table exists. Otherwise, EXISTS(*n*) returns FALSE. For more information, see "Using PL/SQL Table Attributes" on page 4 – 8.
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. For more information, see "Using Host Variables" on page 9 – 7.
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. For more information, see "Using Indicator Variables" on page 9 – 12.
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.
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 always 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!'` yields `'Good morning!'`
	The next example shows that nulls have no effect on the result of a concatenation:
	`'suit' || NULL || 'case'` yields `'suitcase'`
	A string zero characters in length ('') is called a null string and 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 – 20.

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.

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.

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'
```

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

Description

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 – 9.

Syntax

fetch_statement ::=

```
FETCH {  cursor_name
       | cursor_variable_name
       | :host_cursor_variable_name}
   INTO {variable_name[, variable_name]... | record_name};
```

Keyword and Parameter Description

cursor_name	This identifies an explicit cursor previously declared within the current scope.
cursor_variable_name	This identifies a PL/SQL cursor variable (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.
variable_name[, variable_name]...	This identifies a list of previously declared scalar variables into which column 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 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.

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 – 33.

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);
   num    INTEGER(2) := 2;
   CURSOR emp_cur IS SELECT num*sal FROM emp;
BEGIN
   OPEN emp_cur;  -- num equals 2 here
   LOOP
      FETCH emp_cur INTO my_sal;
      EXIT WHEN emp_cur%NOTFOUND;
         -- process the data
      num := num + 1;  -- does not affect next FETCH; sal will
                       -- be multiplied by 2
   END LOOP;
   CLOSE emp_cur;
END;
```

In this example, each retrieved value equals 2 * *sal*, even though *num* is incremented after each fetch. To change the result set or the values of variables in the query, you must close and reopen the cursor with the variables set to their new values.

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;
   /* process the data. */
}
```

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.

```
/* Exit loop when done fetching. */
EXEC SQL WHENEVER NOTFOUND DO break;
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;
   /* process the data. */
}
```

Related Topics

CLOSE Statement, Cursors, Cursor Variables, LOOP Statements, OPEN Statement, OPEN–FOR Statement

Functions

Description

A function is a named program unit that takes parameters and returns a computed value. For more information, see “Functions” on page 7 – 5.

A function has two parts: the specification and the body. The function specification 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 objects 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.

Syntax

function_specification ::=

```
FUNCTION function_name [(parameter_declaration[,
  parameter_declaration]...)]
RETURN return_type;
```

function_body ::=

```
FUNCTION function_name [(parameter_declaration[,
  parameter_declaration]...)]
RETURN return_type IS
  [[object_declaration [object_declaration] ...]
   [subprogram_declaration [subprogram_declaration] ...]]
BEGIN
    seq_of_statements
[EXCEPTION
    exception_handler [exception_handler] ...]
END [function_name];
```

parameter_declaration ::=

```
parameter_name [IN | OUT | IN OUT]
   {  cursor_name%ROWTYPE
    | cursor_variable_name%TYPE
    | plsql_table_name%TYPE
    | record_name%TYPE
    | scalar_type_name
    | table_name%ROWTYPE
    | table_name.column_name%TYPE
    | variable_name%TYPE} [{:= | DEFAULT} expression]
```

return_type ::=

```
{  cursor_name%ROWTYPE
 | cursor_variable_name%ROWTYPE
 | plsql_table_name%TYPE
 | record_name%TYPE
 | scalar_type_name
 | table_name%ROWTYPE
 | table_name.column_name%TYPE
 | variable_name%TYPE}
```

object_declaration ::=

```
{  constant_declaration
 | cursor_declaration
 | cursor_variable_declaration
 | exception_declaration
 | plsql_table_declaration
 | record_declaration
 | variable_declaration}
```

subprogram_declaration ::=

```
{function_declaration | procedure_declaration}
```

Keyword and Parameter Description

function_name — This identifies a user–defined function. For naming conventions, see "Identifiers" on page 2 – 4.

parameter_name — This identifies a formal parameter, which is a variable declared in a function specification 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.
:= | 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. For the syntax of *expression*, see “Expressions” on page 10 – 41. 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.
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.
plsql_table_name	This identifies a PL/SQL table previously declared within the current scope.
record_name	This identifies a user–defined record previously declared within the current scope.
scalar_type_name	This identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2, which must be specified without constraints. For more information, see “Datatypes” on page 2 – 10.
table_name	This identifies a database table (or view) that must be accessible when the declaration is elaborated.
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 PL/SQL variable previously declared 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 or 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 field, record, PL/SQL table, database column, or variable.
constant_declaration	This construct declares a constant. For the syntax of *constant_declaration*, see "Constants and Variables" on page 10 – 16.
cursor_declaration	This construct declares an explicit cursor. For the syntax of *cursor_declaration*, see "Cursors" on page 10 – 23.
cursor_variable_ declaration	This construct declares a cursor variable. For the syntax of *cursor_variable_declaration*, see "Cursor Variables" on page 10 – 27.
exception_declaration	This construct declares an exception. For the syntax of *exception_declaration*, see "Exceptions" on page 10 – 36.
plsql_table_declaration	This construct declares a PL/SQL table. For the syntax of *plsql_table_declaration*, see "PL/SQL Tables" on page 10 – 82.
record_declaration	This construct declares a user–defined record. For the syntax of *record_declaration*, see "Records" on page 10 – 93.
variable_declaration	This construct declares a variable. For the syntax of *variable_declaration*, see "Constants and Variables" on page 10 – 16.
function_declaration	This construct declares a nested function.
procedure_declaration	This construct declares a procedure. For the syntax of *procedure_declaration*, see "Procedures" on page 10 – 87.
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 10 – 36.

Usage Notes

Every function must contain at least one RETURN statement. Otherwise, PL/SQL raises the predefined exception PROGRAM_ERROR at run time.

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);
```

To be callable from SQL expressions, a stored function must obey certain rules meant to control side effects. For standalone functions, Oracle can enforce these rules by checking the function body. However, the body of a packaged function is hidden. So, for packaged functions, you must use the pragma RESTRICT_REFERENCES to enforce the rules. For more information, see "Calling Stored Functions from SQL Expressions" in *Oracle7 Server Application Developer's Guide.*

You can write the function specification and body as a unit. Or, you can separate the function specification 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 specifications in the package specification. However, such functions can be called only from inside the package.

Inside a function, an IN parameter acts like a constant. Therefore, it cannot be assigned a value. An OUT parameter acts like an uninitialized variable. So, its value cannot be assigned to another variable or reassigned to itself. 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. For summary information about the parameter modes, see Table 7 – 1 on page 7 – 15.

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. It is poor programming practice to have a function return multiple values. Also, functions should be free from side effects, which change the values of variables not local to the subprogram. Thus, a function should not change the values of its actual parameters.

Functions can be defined using any Oracle tool that supports PL/SQL. However, to become available for general use, functions must be CREATEd and stored in an Oracle database. You can issue the CREATE FUNCTION statement interactively from SQL*Plus or Server Manager. For the full syntax of the CREATE FUNCTION statement, see *Oracle7 Server SQL Reference.*

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

Packages, PL/SQL Tables, Procedures, Records

GOTO Statement

Description

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 – 14.

Syntax

```
label_declaration ::=

<<label_name>>

goto_statement ::=

GOTO label_name;
```

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;
END;
```

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:

```
BEGIN
   ...
   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
END;
```

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

```
BEGIN
   ...
   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;
END;
```

For more examples of legal and illegal GOTO statements, see “GOTO Statement” on page 3 – 14.

IF Statement

Description

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" on page 3 – 2.

Syntax

if_statement ::=

```
IF boolean_expression THEN
   seq_of_statements
[ELSIF boolean_expression THEN
   seq_of_statements
[ELSIF boolean_expression THEN
   seq_of_statements] ...]
[ELSE
   seq_of_statements]
END IF;
```

Keyword and Parameter Description

boolean_expression — This is an expression that yields the Boolean value TRUE, FALSE, or NULL. It is associated with a sequence of statements, which is executed only if the expression yields TRUE. For the syntax of *boolean_expression*, see "Expressions" on page 10 – 41.

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

Description

The INSERT statement adds new rows of data to a specified database table or view. For a full description of the INSERT statement, see *Oracle7 Server SQL Reference.*

Syntax

```
insert_statement ::=

INSERT INTO {table_reference | (subquery)}
   [(column_name[, column_name]...)]
   {VALUES (sql_expression[, sql_expression]...) | subquery};

table_reference ::=

[schema_name.]{table_name | view_name}[@dblink_name]
```

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.

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 *Oracle7 Server 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. For more information, see “Datatypes” on page 2 – 10.

subquery

This is a select statement that provides a value or set of values to the INSERT statement. The syntax of *subquery* is like the syntax of *select_into_statement* defined in "SELECT INTO Statement" on page 10 – 104, except that *subquery* cannot have an INTO clause.

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.

Usage Notes

All 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

Description

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}

integer ::=

digit[digit]...

real_number ::=

{  integer[.integer]
 | integer.
 | .integer}[{E | e}[+ | –]integer]

character_literal ::=

{'character' | ''''}

string_literal ::=

'{character[character]... | ''['']...}'

boolean_literal ::=

{TRUE | FALSE | NULL}
```

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	This is a predefined Boolean value.
NULL	This is a predefined non–value, which stands for a missing, unknown, or inapplicable value.

Usage Notes

Two kinds of numeric literals can be used in arithmetic expressions: integers and reals. Numeric literals must be separated by punctuation. Space characters 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 'White' and 'White ' are different. How a string literal compares to a variable does *not* depend on the variable; trailing blanks in a literal are never trimmed.

Unlike the non–value NULL, 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

Description

The LOCK TABLE statement lets you lock entire database tables in a specified lock mode so that you can share or deny access to tables while maintaining their integrity. For more information, see "Using LOCK TABLE" on page 5 – 46.

Syntax

```
lock_table_statement ::=

LOCK TABLE table_reference[, table_reference]...
   IN lock_mode MODE [NOWAIT];

table_reference ::=

[schema_name.]{table_name | view_name}[@dblink_name]
```

Keyword and Parameter Description

table_reference — This identifies a table or view that must be accessible when you execute the LOCK TABLE statement.

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

Description

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" on page 3 – 6.

Syntax

basic_loop_statement ::=

```
[<<label_name>>]
LOOP
   seq_of_statements
END LOOP [label_name];
```

while_loop_statement ::=

```
[<<label_name>>]
WHILE boolean_expression
LOOP
   seq_of_statements
END LOOP [label_name];
```

for_loop_statement ::=

```
[<<label_name>>]
FOR index_name IN [REVERSE] lower_bound..upper_bound
LOOP
   seq_of_statements
END LOOP [label_name];
```

cursor_for_loop_statement ::=

```
[<<label_name>>]
FOR record_name IN
 {  cursor_name [(cursor_parameter_name[,
      cursor_parameter_name]...)]
  | (select_statement)}
LOOP
   seq_of_statements
END LOOP [label_name];
```

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* 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 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 10 – 41.

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, are discussed later.) 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. After each iteration, the loop index is decremented.

In this case, 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. An example follows:

```
FOR i IN REVERSE 1..10 LOOP  -- i starts at 10, ends at 1
   -- statements here execute 10 times
END LOOP;
```

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. A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be IN parameters. For the syntax of *cursor_parameter_declaration*, see "Cursors" on page 10 – 23.

select_statement — This is a query associated with an internal cursor unavailable to you. 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.

The syntax of *select_statement* is like the syntax of *select_into_ statement* defined in "SELECT INTO Statement" on page 10 – 104, except that *select_statement* cannot have an INTO clause.

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. For more information, see "EXIT Statement" on page 10 – 39.

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

Description

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 – 17.

Syntax

```
null_statement ::=

NULL;
```

Usage Notes

The NULL statement improves readability by making the meaning and action of conditional statements clear. It tells readers that the associated alternative 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.

Do not confuse the NULL statement with the Boolean non–value NULL; they 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 ZERO_DIVIDE THEN
      ROLLBACK;
   ...
   WHEN OTHERS THEN
      NULL;
END;
```

OPEN Statement

Description

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 – 9.

Syntax

open_statement ::=

```
OPEN cursor_name [(cursor_parameter_name[,
  cursor_parameter_name]...)];
```

Keyword and Parameter Description

cursor_name — This identifies an explicit cursor previously declared within the current scope and not currently open.

cursor_parameter_name — This identifies a cursor parameter; that is, a variable declared as the formal parameter of a cursor. A cursor parameter can appear in a query wherever a constant can appear. For the syntax of *cursor_parameter_declaration*, see "Cursors" on page 10 – 23.

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 bumped 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. The FETCH statement retrieves the rows. With a FOR UPDATE cursor, the rows are locked when the cursor is opened.

If a cursor is currently open, you cannot use its name in a cursor FOR loop.

If formal parameters are declared, actual parameters must be passed to the cursor. 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.

The formal parameters of a cursor must be IN parameters. Therefore, they cannot return values to actual parameters.

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 – 12.

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);
OPEN emp_cur('TRUSDALE', my_comm);
```

In the last example, an actual parameter in the OPEN statement has the same name as its corresponding formal parameter in the cursor declaration. To avoid confusion, use unique identifiers.

Related Topics

CLOSE Statement, Cursors, FETCH Statement, LOOP Statements

OPEN–FOR Statement

Description

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 – 17.

Syntax

open–for_statement ::=

```
OPEN {cursor_variable_name | :host_cursor_variable_name}
   FOR select_statement;
```

Keyword and Parameter Description

cursor_variable_name — This identifies a cursor variable (or parameter) previously declared within the current scope.

host_cursor_variable_name — This identifies a cursor variable previously declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

select_statement — This is a query associated with *cursor_variable*, which returns a set of values. The query can reference bind variables and PL/SQL variables, parameters, and functions 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 10 – 104, 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 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

Packages

Description

A package is a database object that groups logically related PL/SQL types, objects, and subprograms. Packages have two parts: a specification and a body. For more information, see Chapter 8.

Syntax

package_specification ::=

```
PACKAGE package_name IS
   {object_declaration | spec_declaration}
   [{object_declaration | spec_declaration}]...
END [package_name];
```

package_body ::=

```
PACKAGE BODY package_name IS
   [[object_declaration [object_declaration] ...]
    [body_declaration [body_declaration] ...]]
[BEGIN
   seq_of_statements]
END [package_name];
```

object_declaration ::=

```
{  constant_declaration
 | cursor_declaration
 | exception_declaration
 | plsql_table_declaration
 | record_declaration
 | variable_declaration}
```

spec_declaration ::=

```
{  cursor_specification
 | function_specification
 | procedure_specification}
```

body_declaration ::=

```
{  cursor_body
 | function_body
 | procedure_body}
```

Keyword and Parameter Description

package_name	This identifies a package. For naming conventions, see "Identifiers" on page 2 – 4.
constant_declaration	This construct declares a constant. For the syntax of *constant_declaration*, see "Constants and Variables" on page 10 – 16.
cursor_declaration	This construct, which cannot contain a RETURN clause, declares an explicit cursor. For the syntax of *cursor_declaration*, see "Cursors" on page 10 – 23.
exception_declaration	This construct declares an exception. For the syntax of *exception_declaration*, see "Exceptions" on page 10 – 36.
plsql_table_declaration	This construct declares a PL / SQL table. For the syntax of *plsql_table_declaration*, see "PL / SQL Tables" on page 10 – 82.
record_declaration	This construct declares a user–defined record. For the syntax of *record_declaration*, see "Records" on page 10 – 93.
variable_declaration	This construct declares a variable. For the syntax of *variable_declaration*, see "Constants and Variables" on page 10 – 16.
cursor_specification	This construct declares the interface to an explicit cursor. For the syntax of *cursor_specification*, see "Cursors" on page 10 – 23.
function_specification	This construct declares the interface to a function. For the syntax of *function_specification*, see "Functions" on page 10 – 51.
procedure_specification	This construct declares the interface to a procedure. For the syntax of *procedure_specification*, see "Procedures" on page 10 – 87.
cursor_body	This construct defines the underlying implementation of an explicit cursor. For the syntax of *cursor_body*, see "Cursors" on page 10 – 23.
procedure_body	This construct defines the underlying implementation of a procedure. For the syntax of *procedure_body*, see "Procedures" on page 10 – 87.
function_body	This construct defines the underlying implementation of a function. For the syntax of *function_body*, see "Functions" on page 10 – 51.

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 interactively from SQL*Plus or Server Manager and from an Oracle Precompiler or OCI host program. For the full syntax of the CREATE PACKAGE statement, see *Oracle7 Server SQL Reference*.

Most packages have a specification and a body. The specification is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the specification.

Only subprograms and cursors have an underlying implementation (definition). So, if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary. However, the body can still be used to initialize objects declared in the specification, 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 specification without its body. Once the specification 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 specification) to the package body. That means you need not recompile calling programs.

Cursors and subprograms declared in a package specification must be defined in the package body. Other program objects declared in the package specification cannot be redeclared in the package body.

To match subprogram specifications 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

Cursors, Exceptions, Functions, PL/SQL Tables, Procedures, Records

PL/SQL Table Attributes

Description

Every PL/SQL table has the attributes EXISTS, COUNT, FIRST, LAST, PRIOR, NEXT, and DELETE. They make PL/SQL tables easier to use and your applications easier to maintain.

The attributes EXISTS, PRIOR, NEXT, and DELETE take parameters. Each parameter must be an expression that yields a BINARY_INTEGER value or a value implicitly convertible to that datatype.

DELETE acts like a procedure, which is called as a statement. The other PL/SQL table attributes act like a function, which is called as part of an expression. For more information, see "Using PL/SQL Table Attributes" on page 4 – 8.

Syntax

`plsql_table_attribute ::=`

```
plsql_table_name{  .COUNT
                 | .DELETE[(index[, index])]
                 | .EXISTS(index)
                 | .FIRST
                 | .LAST
                 | .NEXT(index)
                 | .PRIOR(index)}
```

Keyword and Parameter Description

plsql_table_name — This identifies a PL/SQL table previously declared within the current scope.

COUNT — This is a PL/SQL table attribute, which can be appended to the name of a PL/SQL table. COUNT returns the number of elements that a PL/SQL table contains.

DELETE — This is a PL/SQL table attribute, which can be appended to the name of a PL/SQL table. This attribute has three forms. DELETE removes all elements from a PL/SQL table. DELETE(*n*) removes the *n*th element. If *n* is null, DELETE(*n*) does nothing. DELETE(*m*, *n*) removes all elements in the range *m* .. *n*. If *m* is larger than *n* or if *m* or *n* is null, DELETE(*m*, *n*) does nothing.

index	This is a numeric expression that must yield a value of type BINARY_INTEGER or a value implicitly convertible to that datatype. For more information, see "Datatype Conversion" on page 2 – 20.
EXISTS	This is a PL/SQL table attribute, which can be appended to the name of a PL/SQL table. EXISTS(*n*) returns TRUE if the *n*th element in a PL/SQL table exists. Otherwise, EXISTS(*n*) returns FALSE.
FIRST, LAST	These are PL/SQL table attributes, which can be appended to the name of a PL/SQL table. FIRST and LAST return the first and last (smallest and largest) index numbers in a PL/SQL table. If the PL/SQL table is empty, FIRST and LAST return nulls. If the PL/SQL table contains only one element, FIRST and LAST return the same index number.
NEXT, PRIOR	These are PL/SQL table attributes, which can be appended to the name of a PL/SQL table. NEXT(*n*) returns the index number that succeeds index *n* in a PL/SQL table. PRIOR(*n*) returns the index number that precedes index *n*. If *n* has no successor, NEXT(*n*) returns a null. Likewise, if *n* has no predecessor, PRIOR(*n*) returns a null.

Usage Notes

Currently, you cannot use PL/SQL table attributes in a SQL statement. If you try, you get a compilation error.

DELETE lets you free the resources held by a PL/SQL table. DELETE(*n*) and DELETE(*m*, *n*) let you prune a PL/SQL table. If an element to be deleted does not exist, DELETE simply skips it; no exception is raised.

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

You can use EXISTS to avoid the exception NO_DATA_FOUND, which is raised when you reference a nonexistent element.

You can use PRIOR or NEXT to traverse PL/SQL tables from which some elements have been deleted.

Examples

In the following example, you delete elements 20 through 30 from a PL/SQL table:

```
ename_tab.DELETE(20, 30);
```

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 emp_tab.FIRST .. emp_tab.LAST LOOP
   ...
END LOOP;
```

In the following example, PL/SQL executes the assignment statement only if the element *sal_tab(i)* exists:

```
IF sal_tab.EXISTS(i) THEN
   sal_tab(i) := sal_tab(i) + 500;
ELSE
   RAISE salary_missing;
END IF;
```

You can use PRIOR or NEXT to traverse PL/SQL tables from which some elements have been deleted, as the following generic example shows:

```
DECLARE
   ...
   i BINARY_INTEGER;
BEGIN
   ..
   i := any_tab.FIRST;  -- get index of first element
   WHILE i IS NOT NULL LOOP
      ...  -- process any_tab(i)
      i := any_tab.NEXT(i);  -- get index of next element
   END LOOP;
END;
```

Related Topics

PL/SQL Tables

PL/SQL Tables

Description

PL/SQL tables are objects of type TABLE, which are modelled as (but not the same as) database tables. PL/SQL tables use a primary key to give you array–like access to rows. Like an array, a PL/SQL table is an ordered collection of elements of the same type. Each element has a unique index number that determines its position in the ordered collection.

However, PL/SQL tables differ from arrays in two important ways. First, arrays have fixed lower and upper bounds, but PL/SQL tables are unbounded. So, the size of a PL/SQL table can increase dynamically. Second, arrays require consecutive index numbers, but PL/SQL tables do not. So, a PL/SQL table can be indexed by any series of integers. For more information, see "PL/SQL Tables" on page 4 – 2.

To create PL/SQL tables, you must take two steps. First, you define a TABLE type, then declare PL/SQL tables of that type.

Syntax

table_type_definition ::=

```
TYPE table_type_name IS TABLE OF
   {  cursor_name%ROWTYPE
    | record_type_name
    | record_name%TYPE
    | scalar_type_name
    | table_name%ROWTYPE
    | table_name.column_name%TYPE
    | variable_name%TYPE} [NOT NULL] INDEX BY BINARY_INTEGER;
```

plsql_table_declaration ::=

```
plsql_table_name table_type_name;
```

Keyword and Parameter Description

table_type_name — This identifies a user–defined type specifier, which is used in subsequent declarations of PL/SQL tables.

cursor_name — This identifies an explicit cursor previously declared within the current scope.

record_type_name — This identifies a RECORD type previously defined within the current scope. For more information, see "User–Defined Records" on page 4 – 19.

record_name	This identifies a user–defined record previously declared within the current scope.
scalar_type_name	This identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2, which must be specified without constraints. For more information, see “Datatypes” on page 2 – 10.
table_name	This identifies a database table (or view) that must be accessible when the declaration is elaborated.
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 PL / SQL variable previously declared 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 record, database column, or variable.
INDEX BY BINARY INTEGER	The index of a PL / SQL table must have datatype BINARY_INTEGER, which can represent signed integers. The magnitude range of a BINARY_INTEGER value is –2147483647 .. 2147483647.
plsql_table_name	This identifies an entire PL / SQL table.

Usage Notes

You can define TABLE types in the declarative part of any block, subprogram, or package. To specify the element type, you can use %TYPE or %ROWTYPE.

A PL / SQL table is unbounded; its index can include any BINARY_ INTEGER value. So, you cannot initialize a PL / SQL table in its declaration. For example, the following declaration is illegal:

```
sal_tab SalTabTyp := (1500, 2750, 2000, 950, 1800);  -- illegal
```

The INDEX BY clause must specify datatype BINARY_INTEGER, which has a magnitude range of –2147483647 .. 2147483647. If the element type is a record type, every field in the record must have a scalar datatype such as CHAR, DATE, or NUMBER.

You can declare PL / SQL tables as the formal parameters of functions and procedures. That way, you can pass PL / SQL tables to stored subprograms and from one subprogram to another.

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

Every PL/SQL table has the attributes EXISTS, COUNT, FIRST, LAST, PRIOR, NEXT, and DELETE. They make PL/SQL tables easier to use and your applications easier to maintain. For more information, see "Using PL/SQL Table Attributes" on page 4 – 8.

The first reference to an element in a PL/SQL table must be an assignment. Until an element is assigned a value, it does not exist. If you reference a nonexistent element, PL/SQL raises the predefined exception NO_DATA_FOUND.

To reference elements in a PL/SQL table, you specify an index number using the following syntax:

```
plsql_table_name(index)
```

When calling a function that returns a PL/SQL table, you use the following syntax to reference elements in the table:

```
function_name(parameters)(index)
```

If the function result is a PL/SQL table of records, you use the following syntax to reference fields in a record:

```
function_name(parameters)(index).field_name
```

Currently, you cannot use the syntax above to call a parameterless function because PL/SQL does not allow empty parameter lists. That is, the following syntax is illegal:

```
function_name()(index)  -- illegal; empty parameter list
```

Instead, declare a local PL/SQL table to which you can assign the function result, then reference the PL/SQL table directly.

You can retrieve Oracle data into a PL/SQL table in three ways: the SELECT INTO statement lets you select a single row of data; the FETCH statement or a cursor FOR loop lets you fetch multiple rows.

Using the SELECT INTO statement, you can select a column entry into a scalar element. Or, you can select an entire row into a record element. Using the FETCH statement or a cursor FOR loop, you can fetch an entire column of Oracle data into a PL/SQL table of scalars. Or, you can fetch an entire table of Oracle data into a PL/SQL table of records.

You cannot reference record variables in the VALUES clause. So, you cannot insert entire records from a PL/SQL table of records into rows in a database table.

With the Oracle Call Interface (OCI) or the Oracle Precompilers, you can bind host arrays to PL/SQL tables declared as the formal parameters of a subprogram. That allows you to pass host arrays to stored functions and procedures.

You can use a BINARY_INTEGER variable or compatible host variable to index the host arrays. Given the array subscript range *m* .. *n*, the corresponding PL/SQL table index range is always 1 .. *n* – *m* + 1. For example, if the array subscript range is 5 .. 10, the corresponding PL/SQL table index range is 1 .. (10 – 5 + 1) or 1 .. 6.

Examples

In the following example, you define a TABLE type named *SalTabTyp*:

```
DECLARE
   TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
      INDEX BY BINARY_INTEGER;
```

Once you define type *SalTabTyp*, you can declare PL/SQL tables of that type, as follows:

```
sal_tab SalTabTyp;
```

The identifier *sal_tab* represents an entire PL/SQL table.

In the next example, you assign the sum of variables *salary* and *increase* to the tenth row in PL/SQL table *sal_tab*:

```
sal_tab(10) := salary * increase;
```

In the following example, you select a row from the database table *dept* into a record stored by the first element of the PL/SQL table *dept_tab*:

```
DECLARE
   TYPE DeptTabTyp IS TABLE OF dept%ROWTYPE
      INDEX BY BINARY_INTEGER;
   dept_tab DeptTabTyp;
BEGIN
   /* Select entire row into record stored by first element. */
   SELECT * INTO dept_tab(1) FROM dept WHERE deptno = 10;
   IF dept_tab(1).dname = 'ACCOUNTING' THEN
      ...
   END IF;
   ...
END;
```

In the final example, you fetch rows from a cursor into the PL/SQL table of records *emp_tab*:

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE
      INDEX BY BINARY_INTEGER;
   emp_tab EmpTabTyp;
   i BINARY_INTEGER := 0;
   CURSOR c1 IS SELECT * FROM emp;
BEGIN
   OPEN c1;
   LOOP
      i := i + 1;
      /* Fetch entire row into record stored by ith element. */
      FETCH c1 INTO emp_tab(i);
      EXIT WHEN c1%NOTFOUND;
      -- process data record
   END LOOP;
   CLOSE c1;
END;
```

Related Topics Functions, PL/SQL Table Attributes, Procedures, Records

Procedures

Description

A procedure is a named PL/SQL block, which can take parameters and be invoked. Generally, you use a procedure to perform an action. For more information, see “Procedures” on page 7 – 3.

A procedure has two parts: the specification and the body. The procedure specification 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 objects 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.

Syntax

procedure_specification ::=

```
PROCEDURE procedure_name (parameter_declaration[,
  parameter_declaration]...)];
```

procedure_body ::=

```
PROCEDURE procedure_name [(parameter_declaration[,
  parameter_declaration]...)] IS
   [[object_declaration [object_declaration] ...]
    [subprogram_declaration [subprogram_declaration] ...]]
BEGIN
    seq_of_statements
[EXCEPTION
    exception_handler [exception_handler] ...]
END [procedure_name];
```

parameter_declaration ::=

```
parameter_name [IN | OUT | IN OUT]
   {  cursor_name%ROWTYPE
    | cursor_variable_name%TYPE
    | plsql_table_name%TYPE
    | record_name%TYPE
    | scalar_type_name
    | table_name%ROWTYPE
    | table_name.column_name%TYPE
    | variable_name%TYPE} [{:= | DEFAULT} expression]
```

object_declaration ::=

```
{  constant_declaration
 | cursor_declaration
 | cursor_variable_declaration
 | exception_declaration
 | plsql_table_declaration
 | record_declaration
 | variable_declaration}
```

subprogram_declaration ::=

```
{function_declaration | procedure_declaration}
```

procedure_name	This identifies a user–defined procedure. For naming conventions, see "Identifiers" on page 2 – 4.
parameter_name	This identifies a formal parameter, which is a variable declared in a procedure specification 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.
:= | 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. For the syntax of *expression*, see "Expressions" on page 10 – 41. When the declaration is elaborated, the value of *expression* is assigned to the parameter. The value and the parameter must have compatible datatypes.

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.
plsql_table_name	This identifies a PL/SQL table previously declared within the current scope.
record_name	This identifies a user–defined record previously declared within the current scope.
scalar_type_name	This identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2, which must be specified without constraints. For more information, see “Datatypes” on page 2 – 10.
table_name	This identifies a database table (or view) that must be accessible when the declaration is elaborated.
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 PL/SQL variable previously declared 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 field, record, PL/SQL table, database column, or variable.
constant_declaration	This construct declares a constant. For the syntax of *constant_declaration*, see “Constants and Variables” on page 10 – 16.
cursor_declaration	This construct declares an explicit cursor. For the syntax of *cursor_declaration*, see “Cursors” on page 10 – 23.
cursor_variable_ declaration	This construct declares a cursor variable. For the syntax of *cursor_variable_declaration*, see “Cursor Variables” on page 10 – 27.
exception_declaration	This construct declares an exception. For the syntax of *exception_declaration*, see “Exceptions” on page 10 – 36.

plsql_table_declaration — This construct declares a PL/SQL table. For the syntax of *plsql_table_declaration*, see "PL/SQL Tables" on page 10 – 82.

record_declaration — This construct declares a user–defined record. For the syntax of *record_declaration*, see "Records" on page 10 – 93.

variable_declaration — This construct declares a variable. For the syntax of *variable_declaration*, see "Constants and Variables" on page 10 – 16.

function_declaration — This construct declares a nested function.

procedure_declaration — This construct declares a procedure. For the syntax of *procedure_declaration*, see "Procedures" on page 10 – 87.

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 10 – 36.

Usage Notes

At least one statement must appear in the executable part of a procedure. The NULL statement meets this requirement.

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. Therefore, it cannot be assigned a value. An OUT parameter acts like an uninitialized variable. So, its value cannot be assigned to another variable or reassigned to itself. 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. For summary information about the parameter modes, see Table 7 – 1 on page 7 – 15.

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.

Unlike OUT and IN OUT parameters, IN parameters can be initialized to default values. For more information, see "Parameter Default Values" on page 7 – 15.

You can write the procedure specification and body as a unit. Or, you can separate the procedure specification 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 specifications in the package specification. 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 CREATEd and stored in an Oracle database. You can issue the CREATE PROCEDURE statement interactively from SQL*Plus or Server Manager. For the full syntax of the CREATE PROCEDURE statement, see *Oracle7 Server SQL Reference.*

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

Functions, Packages, PL/SQL Tables, Records

RAISE Statement

Description

The RAISE statement stops normal execution of a PL/SQL block or subprogram and transfers control to the appropriate exception handler. For more information, see "User–Defined Exceptions" on page 6 – 7.

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.

Syntax

```
raise_statement ::=
RAISE [exception_name];
```

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 – 5.

Usage Notes

PL/SQL blocks and subprograms should RAISE an exception only when an error makes it impractical 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

Description

Records are objects of type RECORD. Records have uniquely named fields that can store data values of different types. For more information, see “User–Defined Records” on page 4 – 19.

To create records, you must take two steps. First, you define a RECORD type, then declare user–defined records of that type.

Syntax

record_type_definition ::=

```
TYPE record_type_name IS RECORD (field_declaration[,
  field_declaration]...);
```

record_declaration ::=

```
record_name record_type_name;
```

field_declaration ::=

```
field_name
   {  cursor_name%ROWTYPE
    | cursor_variable_name%TYPE
    | local_field_name%TYPE
    | plsql_table_name%TYPE
    | record_name%TYPE
    | scalar_type_name
    | table_name%ROWTYPE
    | table_name.column_name%TYPE
    | variable_name%TYPE} [[NOT NULL] {:= | DEFAULT} expression]
```

Keyword and Parameter Description

record_type_name — This identifies a user–defined type specifier, which is used in subsequent declarations of records. For naming conventions, see “Identifiers” on page 2 – 4.

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.

local_field_name — This identifies a field previously declared in the same user–defined record definition.

plsql_table_name	This identifies a PL/SQL table previously declared within the current scope.
record_name	This identifies a user–defined record previously declared within the current scope.
scalar_type_name	This identifies a predefined scalar datatype such as BOOLEAN, NUMBER, or VARCHAR2, which must be specified without constraints. For more information, see “Datatypes” on page 2 – 10.
table_name	This identifies a database table (or view) that must be accessible when the declaration is elaborated.
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 PL/SQL variable previously declared 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 field, record, PL/SQL table, database column, or variable.
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.
:= | 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 10 – 41. 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(
      second SMALLINT := 0,
      minute SMALLINT := 0,
      hour   SMALLINT := 0);
```

The next example shows that you can use the %TYPE attribute to specify a field datatype. It also shows that you can add the NOT NULL constraint to any field declaration and so 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 *dname* field in the *dept_rec* record 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, PL/SQL lets you assign one record to another if they have the same datatype. Note, however, that even if their fields match exactly, a user–defined record and a %ROWTYPE record have different types. Second, you can assign a list of column values to a record by using the SELECT or FETCH statement. Just make sure the column names 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(
      minute SMALLINT,
      hour   SMALLINT);
   TYPE MeetingTyp IS RECORD(
      day     DATE,
      time    TimeTyp,     -- nested record
      place   CHAR(20),
      purpose CHAR(50));
```

```
TYPE PartyTyp IS RECORD(
   day  DATE,
   time TimeTyp,        -- nested record
   loc  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 := meeting.time;
```

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 specification. That allows the function to return a user–defined record of the same type. When calling a function that returns a user–defined record, you use the following syntax to reference fields in the record:

```
function_name(parameters).field_name
```

To reference nested fields in a record returned by a function, you use the following syntax:

```
function_name(parameters).field_name.nested_field_name
```

Currently, you cannot use the syntax above to call a parameterless function because PL/SQL does not allow empty parameter lists. That is, the following syntax is illegal:

```
function_name().field_name  -- illegal; empty parameter list
```

You cannot just drop the empty parameter list because the following syntax is also illegal:

```
function_name.field_name  -- illegal; no parameter list
```

Instead, declare a local user–defined record to which you can assign the function result, then reference its fields directly.

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;
   ...
END;
```

Related Topics

Assignment Statement, Functions, PL/SQL Tables, Procedures

RETURN Statement

Description

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 – 7.

Syntax

```
return_statement ::=

RETURN [expression];
```

Keyword and Parameter Description

expression

This is an arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of *expression*, see "Expressions" on page 10 – 41. 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 specification.

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, it is poor programming practice to have multiple exit points in a subprogram.

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

Description

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 – 39.

Syntax

```
rollback_statement ::=

ROLLBACK [WORK] [TO [SAVEPOINT] savepoint_name];
```

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. For more information, see "Unhandled Exceptions" on page 6 – 20.

Related Topics

COMMIT Statement, SAVEPOINT Statement

%ROWTYPE Attribute

Description

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 – 25.

Syntax

rowtype_attribute ::=

```
{cursor_name | cursor_variable_name | table_name}%ROWTYPE
```

Keyword and Parameter Description

cursor_name — This identifies an explicit cursor previously declared within the current scope.

cursor_variable_name — This identifies a PL/SQL cursor variable previously declared within the current scope.

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. In the following example, you declare a record that can store an entire row from the *emp* table:

```
emp_rec emp%ROWTYPE;
```

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 ...
```

You can assign the value of an expression to a specific field, as the following example shows:

```
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

Description

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 – 39.

Syntax

```
savepoint_statement ::=

SAVEPOINT savepoint_name;
```

Keyword and Parameter Description

savepoint_name

This is an undeclared identifier, which marks the current point in the processing of a transaction. For naming conventions, see "Identifiers" on page 2 – 4.

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.

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 can be reused within a transaction. This moves the savepoint from its old position to the current point in the transaction.

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. For more information, see "Unhandled Exceptions" on page 6 – 20.

By default, the number of active savepoints per user process is limited to 5. You or your DBA can raise the limit (up to 255) by increasing the value of the Oracle initialization parameter SAVEPOINTS.

Related Topics

COMMIT Statement, ROLLBACK Statement

SELECT INTO Statement

Description

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 *Oracle7 Server SQL Reference.*

Syntax

select_into_statement ::=

```
SELECT [DISTINCT | ALL] {* | select_item[, select_item]...}
   INTO {variable_name[, variable_name]... | record_name}
   FROM {table_reference | (subquery)} [alias]
          [, {table_reference | (subquery)} [alias]]...
   rest_of_select_statement;
```

select_item ::=

```
{  function_name[(parameter_name[, parameter_name]...)]
 | NULL
 | numeric_literal
 | [schema_name.]{table_name | view_name}.*
 | [[schema_name.]{table_name. | view_name.}]column_name
 | sequence_name.{CURRVAL | NEXTVAL}
 | 'text'} [[AS] alias]
```

table_reference ::=

```
[schema_name.]{table_name | view_name}[@dblink_name]
```

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.

variable_name[, variable_name]... — This identifies a list of previously declared scalar variables into which *select_item* values are 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.

subquery — This is query that provides a value or set of values to the SELECT statement. The syntax of *subquery* is like the syntax of *select_into_statement*, except that *subquery* cannot have an INTO clause.

alias — This is another (usually short) name for the referenced column, table, or view, and can be used in the WHERE clause.

rest_of_select_statement — This is anything that can legally follow the FROM clause in a SELECT statement.

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 group function such as AVG or SUM. (SQL group functions always return a value or a null. So, a SELECT INTO statement that calls a group 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

Description

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 – 44.

Syntax

set_transaction_statement ::=

```
SET TRANSACTION
   {  READ ONLY
    | READ WRITE
    | ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}
    | USE ROLLBACK SEGMENT rollback_segment_name};
```

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 trys 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.

Only the SELECT INTO, OPEN, FETCH, CLOSE, LOCK TABLE, COMMIT, and ROLLBACK statements are allowed in a read–only transaction. For example, including an INSERT statement raises an exception. Also, queries cannot be FOR UPDATE.

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

Description

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. The SQL cursor has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. They give you useful information about the execution of INSERT, UPDATE, DELETE, and SELECT INTO statements. For more information, see "Managing Cursors" on page 5 – 9.

Syntax

```
sql_cursor ::=

SQL{%FOUND | %ISOPEN | %NOTFOUND | %ROWCOUNT}
```

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 group function never raises NO_DATA_FOUND. That is because group 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

Description

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 zero.

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 ::=

```
SQLCODE
```

Usage Notes

You cannot use SQLCODE directly in a SQL statement. For example, the following statement is illegal:

```
INSERT INTO errors VALUES (SQLCODE, ...);
```

Instead, you must assign the value of SQLCODE to a local variable, then use the variable in the SQL statement, as 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.

Related Topics

Exceptions, SQLERRM Function

SQLERRM Function

Description

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 ::=

SQLERRM [(error_number)]
```

Keyword and Parameter Description

error_number — This must be a valid Oracle error number. For a list of Oracle errors, see *Oracle7 Server 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. For example, the following statement is illegal:

```
INSERT INTO errors VALUES (SQLERRM, ...);
```

Instead, you must assign the value of SQLERRM to a local variable, then use the variable in the SQL statement, as 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.

Related Topics Exceptions, SQLCODE Function

%TYPE Attribute

Description

The %TYPE attribute provides the datatype of a field, record, PL/SQL 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 – 24.

Syntax

```
type_attribute ::=

{  cursor_variable_name
 | plsql_table_name
 | record_name
 | record_name.field_name
 | table_name.column_name
 | variable_name}%TYPE
```

Keyword and Parameter Description

cursor_variable_name	This identifies a PL/SQL cursor variable previously declared within the current scope.
plsql_table_name	This identifies a PL/SQL table 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 is the name of a variable previously declared in the same scope. For naming conventions, see "Identifiers" on page 2 – 4.

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 does *not* apply to objects declared using %TYPE.

Related Topics

Constants and Variables, %ROWTYPE Attribute

UPDATE Statement

Description

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 *Oracle7 Server SQL Reference.*

Syntax

update_statement ::=

```
UPDATE {table_reference | (subquery)} [alias]
   SET {  column_name = {sql_expression | (subquery)}
        | (column_name[, column_name]...) = (subquery)}
       [, {  column_name = {sql_expression | (subquery)}
           | (column_name[, column_name]...) = (subquery)}]...
   [WHERE {search_condition | CURRENT OF cursor_name}];
```

table_reference ::=

```
[schema_name.]{table_name | view_name}[@dblink_name]
```

Keyword and Parameter Description

table_reference — This specifies a table or view, which must be accessible when you execute the UPDATE statement, and for which you must have UPDATE privileges.

subquery — This is a select statement that provides a value or set of values to the UPDATE statement. The syntax of *subquery* is like the syntax of *select_into_statement* defined in "SELECT INTO Statement" on page 10 – 104, except that *subquery* cannot have an INTO clause.

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 expression valid in SQL. For more information, see *Oracle7 Server 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 1.1, then the result is assigned to the *sal* column.

```
UPDATE emp SET sal = sal * 1.1;
```

SET column_name = subquery

This clause assigns the value retrieved from the database by *subquery* to the column identified by *column_name*. The subquery must return exactly one row and one column.

SET (column_name[, column_name]...) = subquery

This clause assigns the values retrieved from the database by *subquery* to the columns in the *column_name* list. The subquery must return exactly one row, which includes all the columns listed in parentheses on the left side of the equal sign.

The column values returned by *subquery* are assigned to the columns in the *column_name* list in order. Thus, the first value is assigned to the first column in the *column_name* list, the second value is assigned to the second column in the *column_name* 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.

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 all analysts and clerks in department 20:

```
UPDATE emp SET sal = sal * 1.10
   WHERE (job = 'ANALYST' OR job = 'CLERK') 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';
```

Related Topics

DELETE Statement, FETCH Statement

PART

III Appendices

APPENDIX

A New Features

This appendix surveys the new features in release 2.3 of PL/SQL. Designed to meet your practical needs, these features will help you build effective, reliable applications.

Support for File I/O

The new package UTL_FILE, which is supplied with the Oracle Server, 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 part of the Oracle initialization file.

For more information, see *Oracle7 Server Application Developer's Guide*.

PL/SQL Table Improvements

Now, you can declare PL/SQL tables of records as well as PL/SQL tables of scalars. That means a PL/SQL table can store rows (not just a column) of Oracle data. PL/SQL tables of records make it easy to move collections of data into and out of database tables or between client–side applications and stored subprograms. You can even use PL/SQL tables of records to simulate local database tables.

Also, several new attributes give you previously unavailable information about a PL/SQL table. Attributes are characteristics of an object. Every PL/SQL table has the attributes EXISTS, COUNT, FIRST, LAST, PRIOR, NEXT, and DELETE. They make PL/SQL tables easier to use and your applications easier to maintain.

For example, COUNT returns the number of elements that a PL/SQL table contains. COUNT is useful because the size of a PL/SQL table is unconstrained. Suppose you fetch a column of Oracle data into a PL/SQL table. How many elements does the PL/SQL table contain? COUNT gives you the answer.

To apply the attributes to a PL/SQL table, you use dot notation, as the following example shows:

```
IF ename_tab.COUNT = 50 THEN ...
```

For more information, see "PL/SQL Tables" on page 4 – 2.

Cursor Variable Improvements

Now, 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 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.

Furthermore, now you can define *weak* (nonrestrictive) REF CURSOR types. As the following 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
```

Weak REF CURSOR types are more flexible because PL/SQL lets you associate a weakly typed cursor variable with any query.

Also, now you can apply the cursor attributes %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT to a cursor variable. They return useful information about the execution of a multi–row query.

For more information, see “Using Cursor Variables” on page 5 – 17.

New Fast–Integer Datatype

You can use the new datatype PLS_INTEGER to boost performance. Like BINARY_INTEGER values, PLS_INTEGER values require less storage than NUMBER values. But, PLS_INTEGER operations use machine arithmetic, so they are considerably faster than BINARY_ INTEGER operations, which use library arithmetic.

For more information, see “PLS_INTEGER” on page 2 – 12.

Full Support for Subqueries

Formerly, PL/SQL allowed subqueries only in the SET, VALUES, and WHERE clauses. Now, PL/SQL also allows subqueries in the FROM clause. Among other things, this adds flexibility to your cursor definitions.

For an example, see "Using Subqueries" on page 5 – 13.

New Remote Dependency Mode

Formerly, Oracle used only timestamps to manage remote dependencies among PL/SQL library units (packages and stored subprograms). Whenever a library unit was recompiled, the server timestamped it. At run time, dependent subprograms on a client system or on another server were invalidated because their timestamps were no longer current. Often, the resulting recompilations were needless because the specification of the library unit had not been altered.

Needless recompilations can slow network traffic and affect performance. Furthermore, if a client system has no PL/SQL compiler, invalidated applications cannot be recompiled.

Now, Oracle can use timestamps or signatures to manage remote dependencies. (The *signature* of a subprogram includes its name and the number, datatypes, and modes of its parameters.) When Oracle uses the signature of a remote library unit, dependent subprograms are invalidated only if the signature or specification of the unit was altered. So, dependent subprograms are recompiled only when necessary.

To have Oracle use signatures instead of timestamps, you set the following parameter in the Oracle initialization file:

```
REMOTE_DEPENDENCIES_MODE = SIGNATURE
```

You can reset the parameter dynamically, as the following Pro*C example shows:

```
EXEC SQL ALTER SESSION
   SET REMOTE_DEPENDENCIES_MODE = TIMESTAMP;
```

For more information, see *Oracle7 Server Application Developer's Guide*.

APPENDIX

B Sample Programs

This appendix provides several PL/SQL programs to guide you in writing your own. The sample programs illustrate the following important PL/SQL concepts and features:

- FOR loops
- cursors
- scoping
- batch transaction processing
- embedded PL/SQL
- calling a stored procedure

Running the Samples

The sample programs in this appendix and many others throughout this Guide are available online. Therefore, they are preceded by the following comment:

```
-- available online in file <filename>
```

The list below gives their locations in this Guide and the names of the corresponding online files. However, the exact name and storage location of an online file are system dependent.

Location in Guide	*Online File*
Chapter 1, page 1 – 2	EXAMP1
Chapter 1, page 1 – 8	EXAMP2
Chapter 1, page 1 – 10	EXAMP3
Chapter 2, page 2 – 26	EXAMP4
Chapter 5, page 5 – 35	EXAMP5
Chapter 5, page 5 – 36	EXAMP6
Chapter 5, page 5 – 15	EXAMP7
Chapter 5, page 5 – 16	EXAMP8
Chapter 9, page 9 – 8	EXAMP9
Chapter 9, page 9 – 10	EXAMP10
Chapter 10, page 10 – 11	EXAMP11
Chapter 10, page 10 – 21	EXAMP12
Chapter 10, page 10 – 21	EXAMP13
Chapter 10, page 10 – 22	EXAMP14
Appendix B, page B – 11	SAMPLE1
Appendix B, page B – 12	SAMPLE2
Appendix B, page B – 13	SAMPLE3
Appendix B, page B – 14	SAMPLE4
Appendix B, page B – 18	SAMPLE5
Appendix B, page B – 22	SAMPLE6

Some samples are run interactively from SQL*Plus; others are run 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, supplied with PL/SQL. These scripts can be found in the PL/SQL installation library. Check the Oracle installation or user's guide for your system.

Creating the Tables

Below is a listing of the SQL*Plus script EXAMPBLD. The CREATE statements in this script build the database tables processed by the sample programs. To run the script, invoke SQL*Plus, then issue the following command:

```
SQL> START EXAMPBLD
```

EXAMPBLD Script

```
set compatibility V6
/
drop table accounts
/
create table accounts(
   account_id  number(4) not null,
   bal         number(11,2))
/
create unique index accounts_index on accounts (account_id)
/
drop table action
/
create table action(
   account_id  number(4) not null,
   oper_type   char(1) not null,
   new_value   number(11,2),
   status      char(45),
   time_tag    date not null)
/
drop table bins
/
create table bins(
   bin_num     number(2) not null,
   part_num    number(4),
   amt_in_bin  number(4))
/
drop table data_table
/
```

```
create table data_table(
   exper_num  number(2),
   n1         number(5),
   n2         number(5),
   n3         number(5))
/
drop table emp
/
create table emp(
   empno      number(4) not null,
   ename      char(10),
   job        char(9),
   mgr        number(4),
   hiredate   date,
   sal        number(7,2),
   comm       number(7,2),
   deptno     number(2))
/
drop table inventory
/
create table inventory(
   prod_id     number(5) not null,
   product     char(15),
   quantity    number(5))
/
drop table journal
/
create table journal(
   account_id   number(4) not null,
   action       char(45) not null,
   amount       number(11,2),
   date_tag     date not null)
/
drop table num1_tab
/
create table num1_tab(
   sequence   number(3) not null,
   num        number(4))
/
drop table num2_tab
/
```

```
create table num2_tab(
   sequence   number(3) not null,
   num        number(4))
/
drop table purchase_record
/
create table purchase_record(
   mesg        char(45),
   purch_date  date)
/
drop table ratio
/
create table ratio(
   sample_id  number(3) not null,
   ratio      number)
/
drop table result_table
/
create table result_table(
   sample_id  number(3) not null,
   x          number,
   y          number)
/
drop table sum_tab
/
create table sum_tab(
   sequence   number(3) not null,
   sum        number(5))
/
drop table temp
/
create table temp(
   num_col1   number(9,4),
   num_col2   number(9,4),
   char_col   char(55))
/
create or replace package personnel as
   type charArrayTyp is table of varchar2(10)
      index by binary_integer;
   type numArrayTyp is table of float
      index by binary_integer;
```

```
   procedure get_employees(
      dept_number in     integer,
      batch_size  in     integer,
      found       in out integer,
      done_fetch  out    integer,
      emp_name    out    charArrayTyp,
      job–title   out    charArrayTyp,
      salary      out    numArrayTyp);
end personnel;
/
create or replace package body personnel as
   cursor get_emp (dept_number integer) is
      select ename, job, sal from emp
         where deptno = dept_number;
   procedure get_employees(
      dept_number in     integer,
      batch_size  in     integer,
      found       in out integer,
      done_fetch  out    integer,
      emp_name    out    charArrayTyp,
      job_title   out    charArrayTyp,
      salary      out    numArrayTyp) is
   begin
      if not get_emp%isopen then
         open get_emp(dept_number);
      end if;
      done_fetch := 0;
      found := 0;
      for i in 1..batch_size loop
         fetch get_emp into emp_name(i),
            job_title(i), salary(i);
         if get_emp%notfound then
            close get_emp;
            done_fetch := 1;
            exit;
         else
            found := found + 1;
         end if;
      end loop;
   end get_employees;
end personnel;
/
```

Loading the Data

Below is a listing of the SQL*Plus script EXAMPLOD. The INSERT statements in this script load (or reload) the database tables processed by the sample programs. To run the script, invoke SQL*Plus in the same Oracle account from which you ran EXAMPBLD, then issue the following command:

```
SQL> START EXAMPLOD
```

EXAMPLOD Script

```
delete from accounts
/
insert into accounts values (1,1000.00)
/
insert into accounts values (2,2000.00)
/
insert into accounts values (3,1500.00)
/
insert into accounts values (4,6500.00)
/
insert into accounts values (5,500.00)
/
delete from action
/
insert into action values
   (3,'u',599,null,sysdate)
/
insert into action values
   (6,'i',20099,null,sysdate)
/
insert into action values
   (5,'d',null,null,sysdate)
/
insert into action values
   (7,'u',1599,null,sysdate)
/
insert into action values
   (1,'i',399,null,sysdate)
/
insert into action values
   (9,'d',null,null,sysdate)
/
insert into action values
   (10,'x',null,null,sysdate)
/
delete from bins
/
```

```
insert into bins values (1, 5469, 650)
/
insert into bins values (2, 7243, 450)
/
insert into bins values (3, 5469, 120)
/
insert into bins values (4, 5469, 300)
/
insert into bins values (5, 6085, 415)
/
insert into bins values (6, 5469, 280)
/
insert into bins values (7, 8159, 619)
/
delete from data_table
/
insert into data_table values
   (1, 10, 167, 17)
/
insert into data_table values
   (1, 16, 223, 35)
/
insert into data_table values
   (2, 34, 547, 2)
/
insert into data_table values
   (3, 23, 318, 11)
/
insert into data_table values
   (1, 17, 266, 15)
/
insert into data_table values
   (1, 20, 117, 9)
/
delete from emp
/
insert into emp values
   (7369,'SMITH','CLERK',7902,TO_DATE('12-17-80','MM-DD-YY'),
    800,NULL,20)
/
insert into emp values
   (7499,'ALLEN','SALESMAN',7698,TO_DATE('02-20-81','MM-DD-YY'),
    1600,300,30)
/
insert into emp values
   (7521,'WARD','SALESMAN',7698,TO_DATE('02-22-81','MM-DD-YY'),
    1250,500,30)
/
```

```
insert into emp values
   (7566,'JONES','MANAGER',7839,TO_DATE('04-02-81','MM-DD-YY'),
    2975,NULL,20)
/
insert into emp values
(7654,'MARTIN','SALESMAN',7698,TO_DATE('09-28-81','MM-DD-YY'),
    1250,1400,30)
/
insert into emp values
   (7698,'BLAKE','MANAGER',7839,TO_DATE('05-1-81','MM-DD-YY'),
    2850,NULL,30)
/
insert into emp values
   (7782,'CLARK','MANAGER',7839,TO_DATE('06-9-81','MM-DD-YY'),
    2450,NULL,10)
/
insert into emp values
   (7788,'SCOTT','ANALYST',7566,SYSDATE-85,3000,NULL,20)
/
insert into emp values
   (7839,'KING','PRESIDENT',NULL,TO_DATE('11-17-81','MM-DD-YY'),
    5000,NULL,10)
/
insert into emp values
   (7844,'TURNER','SALESMAN',7698,TO_DATE('09-8-81','MM-DD-YY'),
    1500,0,30)
/
insert into emp values
   (7876,'ADAMS','CLERK',7788,SYSDATE-51,1100,NULL,20)
/
insert into emp values
   (7900,'JAMES','CLERK',7698,TO_DATE('12-3-81','MM-DD-YY'),
    950,NULL,30)
/
insert into emp values
   (7902,'FORD','ANALYST',7566,TO_DATE('12-3-81','MM-DD-YY'),
    3000,NULL,20)
/
insert into emp values
   (7934,'MILLER','CLERK',7782,TO_DATE('01-23-82','MM-DD-YY'),
    1300,NULL,10)
/
delete from inventory
/
insert into inventory values
   ('TENNIS RACKET', 3)
/
```

```
insert into inventory values
   ('GOLF CLUB', 4)
/
insert into inventory values
   ('SOCCER BALL', 2)
/
delete from journal
/
delete from num1_tab
/
insert into num1_tab values (1, 5)
/
insert into num1_tab values (2, 7)
/
insert into num1_tab values (3, 4)
/
insert into num1_tab values (4, 9)
/
delete from num2_tab
/
insert into num2_tab values (1, 15)
/
insert into num2_tab values (2, 19)
/
insert into num2_tab values (3, 27)
/
delete from purchase_record
/
delete from ratio
/
delete from result_table
/
insert into result_table values (130, 70, 87)
/
insert into result_table values (131, 77, 194)
/
insert into result_table values (132, 73, 0)
/
insert into result_table values (133, 81, 98)
/
delete from sum_tab
/
delete from temp
/
commit
```

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

10 records selected.
```

Sample 2. Cursors

The next 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

14 records selected.
```

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

20 records selected.
```

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

7 records selected.
```

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

6 records selected.


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.

7 records selected.
```

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 a procedure named *get_employees*, which is stored in a package named *personnel*. The procedure declares three PL/SQL tables as OUT formal parameters, then fetches a batch of employee data into the PL/SQL tables. The matching actual parameters are host arrays. When the procedure finishes, it automatically assigns all row values in the PL/SQL 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

14 records selected.
```

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
```

APPENDIX

C

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.

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. For example, given the following declaration, the value of *name* includes six trailing blanks, not just one:

```
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

```
name1 VARCHAR2(10) := 'COLES';
name2 VARCHAR2(10) := 'COLEMAN';
```

the following IF condition is true:

```
IF name1 > 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

```
name1 CHAR(5) := 'BELLO';
name2 CHAR(10) := 'BELLO   ';  -- note trailing blanks
```

the following IF condition is true:

```
IF name1 = name2 THEN ...
```

If either or both values in a comparison have 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

```
name1 VARCHAR2(10) := 'DOW';
name2 VARCHAR2(10) := 'DOW   ';  -- note trailing blanks
```

the following IF condition is false:

```
IF name1 = 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

```
name1 VARCHAR2(10) := 'STAUB';
name2 CHAR(10)     := 'STAUB';  -- PL/SQL blank-pads value
```

the following IF condition is false because the value of *name2* includes five trailing blanks:

```
IF name1 = 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 CHAR 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 VARCHAR2 column, Oracle aborts the insert and generates an error.

The same rules apply when updating.

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 CHAR variable, PL/SQL aborts the assignment and raises the predefined exception 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.

The same rules apply when fetching.

Guidelines

In a given execution environment, whether CHAR is equivalent to VARCHAR2 or not is determined by a command or option that sets Oracle Version 6 or Oracle7 compatibility. For example, in the SQL*Plus environment, you issue the SET COMPATIBILITY command, specifying the value V6 or V7 (the default), as follows:

```
SQL> SET COMPATIBILITY V6
```

As the next example shows, in the Oracle Precompiler environment, you enter the runtime option DBMS on the command line, specifying the value V6, V7, or NATIVE (the default). NATIVE specifies the version of Oracle resident on your system, which must be version 6 or later.

```
... DBMS=V6
```

When selecting data over a V7–to–V6 link, use VARCHAR2 variables in the WHERE clause instead of CHAR variables. Otherwise, you might get an *unsupported network datatype* error.

When inserting character values, you can ensure that no trailing blanks are stored by using the RTRIM function, which trims trailing blanks. An example follows:

```
my_empno := 7471;
my_ename := 'LEE   ';  -- note trailing blanks
...
INSERT INTO emp
   VALUES (my_empno, RTRIM(my_ename), ...);  -- inserts 'LEE'
```

APPENDIX

D PL/SQL Wrapper

This appendix shows you how to run the PL/SQL Wrapper, a standalone 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.

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 system prompt using the following syntax:

```
WRAP INAME=input_file [ONAME=output_file]
```

You can use uppercase or lowercase. 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 PL/SQL packages and standalone subprograms:

- CREATE [OR REPLACE] PACKAGE
- CREATE [OR REPLACE] PACKAGE BODY
- CREATE [OR REPLACE] FUNCTION
- CREATE [OR REPLACE] PROCEDURE

All other SQL statements are passed intact to the output file. Comment lines (beginning with REM or – –) are deleted unless they appear in a package or subprogram definition.

A wrapped package or subprogram definition has the form

```
<header> WRAPPED <body>
```

where *header* begins with the reserved word CREATE and ends with the name of the package or subprogram, and *body* is an intermediate form of object code that looks like a random sequence of characters. The keyword WRAPPED tells the PL/SQL compiler that the package or subprogram is wrapped.

The header can contain comments. For example, the Wrapper converts

```
CREATE OR REPLACE PACKAGE
-- Author: J Smith
-- Date:   11/15/94
mypkg AS ...
```

into

```
CREATE OR REPLACE PACKAGE
-- Author: J Smith
-- Date:   11/15/94
mypkg WRAPPED 8c724af33 ...
```

Generally, the output file is much larger than the input file.

Error Detection

If your input file contains syntactic errors, the PL/SQL Wrapper detects and reports them. However, the Wrapper cannot detect semantic errors because it does not resolve external references. That is done at compile time. So, only the PL/SQL compiler can detect semantic errors.

APPENDIX

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. Also, some of these words (marked by an asterisk) are reserved by SQL. So, you should not use them to name database objects such as columns, tables, or indexes.

PL/SQL Reserved Words

ABORT
ACCEPT
ACCESS*
ADD*
ALL*
ALTER*
AND*
ANY*
ARRAY
ARRAYLEN
AS*
ASC*
ASSERT
ASSIGN
AT
AUDIT*
AUTHORIZATION
AVG
BASE_TABLE
BEGIN
BETWEEN*
BINARY_INTEGER
BODY
BOOLEAN
BY*
CASE
CHAR*
CHAR_BASE
CHECK*
CLOSE
CLUSTER*
CLUSTERS
COLAUTH
COLUMN*
COMMENT*
COMMIT
COMPRESS*
CONNECT*
CONSTANT
CRASH
CREATE*
CURRENT*
CURRVAL
CURSOR
DATABASE
DATA_BASE
DATE*
DBA
DEBUGOFF
DEBUGON
DECLARE
DECIMAL*
DEFAULT*
DEFINITION
DELAY
DELETE*
DELTA
DESC*
DIGITS
DISPOSE
DISTINCT*
DO
DROP*
ELSE*
ELSIF
END
ENTRY
EXCEPTION
EXCEPTION_INIT
EXCLUSIVE*
EXISTS*
EXIT
FALSE
FETCH
FILE*
FLOAT*
FOR*
FORM
FROM*
FUNCTION
GENERIC
GOTO
GRANT*
GROUP*
HAVING*
IDENTIFIED*
IF
IMMEDIATE*
IN*
INCREMENT*
INDEX*
INDEXES
INDICATOR
INITIAL*
INSERT*
INTEGER*
INTERFACE
INTERSECT*
INTO*
IS*
LEVEL*
LIKE*
LIMITED
LOCK*
LONG*
LOOP
MAX
MAXEXTENTS*
MIN
MINUS*
MLSLABEL
MOD
MODE*
MODIFY*
NATURAL
NATURALN
NEW
NEXTVAL
NOAUDIT*
NOCOMPRESS*
NOT*

NOWAIT*
NULL*
NUMBER*
NUMBER_BASE
OF*
OFFLINE*
ON*
ONLINE*
OPEN
OPTION*
OR*
ORDER*
OTHERS
OUT
PACKAGE
PARTITION
PCTFREE*
PLS_INTEGER
POSITIVE
POSITIVEN
PRAGMA
PRIOR*
PRIVATE
PRIVILEGES*
PROCEDURE
PUBLIC*
RAISE
RANGE
RAW*
REAL
RECORD
REF
RELEASE
REMR
RENAME*
RESOURCE*
RETURN
REVERSE
REVOKE*
ROLLBACK
ROW*
ROWID*
ROWLABEL*
ROWNUM*
ROWS*
ROWTYPE
RUN
SAVEPOINT
SCHEMA
SELECT*
SEPARATE
SESSION*
SET*
SHARE*
SIZE*
SMALLINT*
SPACE
SQL
SQLCODE
SQLERRM
START*
STATEMENT
STDDEV
SUBTYPE
SUCCESSFUL*
SUM
SYNONYM*
SYSDATE*
TABAUTH
TABLE*
TABLES
TASK
TERMINATE
THEN*
TO*
TRIGGER*
TRUE
TYPE
UID*
UNION*
UNIQUE*
UPDATE*
USE
USER*
VALIDATE*
VALUES*
VARCHAR*
VARCHAR2*
VARIANCE
VIEW*
VIEWS
WHEN
WHENEVER*
WHERE*
WHILE
WITH*
WORK
WRITE
XOR

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Reader's Comment Form

PL/SQL™ User's Guide and Reference
Part No. A32542–1

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 topic, chapter, and page number below:

_____

_____

_____

_____

_____

_____

_____

_____

Please send your comments to:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.
Fax: (415) 506–7200

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

_____

_____

_____

Thank you for helping us improve our documentation.