|  |
| --- |
| , RD Dep. |
| **Oracle PL/SQL** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial Version | Siarhei Kandrashevich | 15-Jan-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Stored procedures, functions and packages 3](#_Toc380733051)

[1.1. Procedures 3](#_Toc380733052)

[1.2. Functions 4](#_Toc380733053)

[1.2.1. Parameters 5](#_Toc380733054)

[1.2.2. Datatype 5](#_Toc380733055)

[1.2.3. Mode 5](#_Toc380733056)

[1.2.4. Default values 6](#_Toc380733057)

[1.2.5. Parameter-passing notations 6](#_Toc380733058)

[1.2.6. Local programs 7](#_Toc380733059)

[1.2.7. Program overloading 8](#_Toc380733060)

[1.2.8. Forward declarations 8](#_Toc380733061)

[1.2.9. Table functions 9](#_Toc380733062)

[1.2.10. Compiling stored PL/SQL programs 10](#_Toc380733063)

[1.2.11. Native compilation of PL/SQL 11](#_Toc380733064)

[1.2.12. Privileges and stored PL/SQL 11](#_Toc380733065)

[1.3. Packages 11](#_Toc380733066)

[1.3.1. Package Structure 12](#_Toc380733067)

[1.3.2. Referencing Package Elements 13](#_Toc380733068)

[1.3.3. Package Data 14](#_Toc380733069)

[1.3.4. SERIALLY\_REUSABLE Pragma 14](#_Toc380733070)

[1.3.5. Package Initialization 14](#_Toc380733071)

[2. Error checking - exception handling 16](#_Toc380733072)

[2.1. Exception Handling 16](#_Toc380733073)

[2.1.1. Declaring Exceptions 16](#_Toc380733074)

[2.1.2. Raising Exceptions 17](#_Toc380733075)

[2.1.3. Scope 18](#_Toc380733076)

[2.1.4. Propagation 18](#_Toc380733077)

[2.1.5. WHEN OTHERS clause 19](#_Toc380733078)

[2.1.6. SQLCODE and SQLERRM 19](#_Toc380733079)

[2.1.7. Exceptions and DML 20](#_Toc380733080)

[3. Triggers 21](#_Toc380733081)

[3.1. Creating Triggers 21](#_Toc380733082)

[3.2. Trigger Predicates 23](#_Toc380733083)

[3.3. DML Events 23](#_Toc380733084)

[3.4. DDL Events 23](#_Toc380733085)

[3.5. Database Events 24](#_Toc380733086)

[4. Cursors in PL/SQL 25](#_Toc380733087)

[4.1. Explicit Cursors 25](#_Toc380733088)

[4.2. Declaring explicit cursors 25](#_Toc380733089)

[4.3. Opening explicit cursors 25](#_Toc380733090)

[4.4. Fetching from explicit cursors 26](#_Toc380733091)

[4.5. Closing explicit cursors 26](#_Toc380733092)

[4.6. Explicit cursor attributes 26](#_Toc380733093)

[4.7. Implicit Cursors 28](#_Toc380733094)

[4.8. SELECT FOR UPDATE clause 29](#_Toc380733095)

[4.9. WHERE CURRENT OF clause 29](#_Toc380733096)

[4.10. Dynamic Cursors 30](#_Toc380733097)

[4.11. Cursor Variables 31](#_Toc380733098)

[4.12. Cursor Expressions 32](#_Toc380733099)

# 

# Stored procedures, functions and packages

PL/SQL allows you to create a variety of named program units, or containers for code. These include:

*Procedure*

A program that executes one or more statements

*Function*

A program that returns a value

*Package*

A container for procedures, functions, and data structures

*Trigger*

A program that executes in response to database changes

*Object type*

Oracle's version of an object-oriented class; object types can contain member procedures and functions

## Procedures

Procedures are program units that execute one or more statements and can receive or return zero or more values through their parameter lists. The syntax of a procedure is:

CREATE [OR REPLACE] PROCEDURE *name*

[ (*parameter* [,*parameter*]) ]

[AUTHID { CURRENT\_USER | DEFINER } ]

[DETERMINISTIC]

{ IS | AS }

*declaration\_section*

BEGIN

*executable\_section*

[EXCEPTION

*exception\_section*]

END [*name*];

A procedure is called as a standalone executable PL/SQL statement:

apply\_discount(new\_company\_id, 0.15);

## Functions

Functions are program units that execute zero or more statements and return a value through the RETURN clause. Functions can also receive or return zero or more values through their parameter lists. The syntax of a function is:

CREATE [OR REPLACE] FUNCTION *name*

[ (*parameter* [,*parameter*]) ]

RETURN *return\_datatype*

[AUTHID { CURRENT\_USER | DEFINER } ]

[DETERMINISTIC]

[PARALLEL\_ENABLE]

[PIPELINED]

[AGGREGATE USING]

{ IS | AS }

[*declaration\_section*]

BEGIN

*executable\_section*

[EXCEPTION

*exception\_section*]

END [*name*];

A function must have at least one RETURN statement in the execution section. The RETURN clause in the function header specifies the datatype of the returned value.

A function can be called anywhere that an expression of the same type can be used. You can call a function:

* In an assignment statement:

sales95 := tot\_sales(1995,'C');

* To set a default value:
* DECLARE
* sales95 NUMBER DEFAULT tot\_sales(1995,'C');

BEGIN

* In a Boolean expression:
* IF tot\_sales(1995,'C') > 10000
* THEN

...

* In a SQL statement:
* SELECT first\_name ,surname
* FROM sellers

WHERE tot\_sales(1995,'C') > 1000;

* As an argument in another program unit's parameter list.

Here, for example, max\_discount is a programmer-defined function and SYSDATE is a built-in function:

apply\_discount(company\_id, max\_discount(SYSDATE));

### Parameters

Procedures, functions, and cursors may have a parameter list. This list contains one or more parameters that allow you to pass information back and forth between the sub-program and the calling program. Each parameter is defined by its name, datatype, mode, and optional default value. The syntax for a parameter is:

*parameter\_name* [*mode*] [NOCOPY] *datatype*

[ { := | DEFAULT } *value*]

### Datatype

The datatype can be any PL/SQL or programmer-defined datatype, but cannot be constrained by a size (NUMBER is valid, NUMBER(10) is not valid). The actual size of the parameter is determined from the calling program or via a %TYPE constraint.

CREATE OR REPLACE PROCEDURE empid\_to\_name

(in\_id emp.emp\_id%TYPE -- Compiles OK.

,out\_last\_name VARCHAR2 -- Compiles OK.

,out\_first\_name VARCHAR2(10) -- Won't compile.

) IS

...

The lengths of out\_last\_name and out\_first\_name are determined by the calling program:

DECLARE

surname VARCHAR2(10);

first\_name VARCHAR2(10);

BEGIN

empid\_to\_name(10, surname, first\_name);

END;

### Mode

The mode of a parameter specifies whether the parameter can be read from or written to, as shown in the following table:

|  |  |  |
| --- | --- | --- |
| **Mode** | **Description** | **Parameter usage** |
| IN | Read-only | The value of the actual parameter can be referenced inside the program, but the parameter cannot be changed. |
| OUT or IN OUT | Read/write | The program can both reference (read) and modify (write) the parameter. |

If the mode is not explicitly defined, it defaults to IN.

OUT parameters are not the same as IN OUT parameters. When running the called program, the runtime engine ignores (sets to NULL) any argument value you supply for an OUT parameter; it preserves the value provided for an IN OUT. If an exception is raised during execution of a procedure or function, assignments made to OUT or IN OUT parameters get rolled back unless the parameter includes the NOCOPY option.

The NOCOPY compiler hint for parameters makes the parameter a call by reference instead of a call by value. Normally, PL/SQL passes IN/OUT parameters by value—a copy of the parameter is created for the sub-program. When parameter items get large, as collections and objects do, the copy can eat memory and slow down processing. NOCOPY directs PL/SQL to pass the parameter by reference, using a pointer to the single copy of the parameter.

The disadvantage of NOCOPY is that when an exception is raised during execution of a program that has modified an OUT or IN OUT parameter, the changes to the actual parameters are not "rolled back" because the parameters were passed by reference instead of being copied.

### Default values

IN parameters can be given default values. If an IN parameter has a default value, then you do not need to supply an argument for that parameter when you call the program unit. It automatically uses the default value. For example:

CREATE OR REPLACE PROCEDURE hire\_employee

(emp\_id IN VARCHAR2

,hire\_date IN DATE := SYSDATE

,company\_id IN NUMBER := 1

)

IS

...

Here are some example calls to the above procedure:

-- Use two default values.

hire\_employee(new\_empno);

-- Use one default value.

hire\_employee(new\_empno,'12-Jan-1999');

-- Use non-trailing default value, named notation.

hire\_employee(emp\_id=>new\_empno, comp\_id=>12);

### Parameter-passing notations

Formal parameters are the names that are declared in the header of a procedure or function. Actual parameters (arguments) are the values or expressions placed in the parameter list when a procedure or function is called.

PL/SQL lets you use either of two styles for passing arguments in parameter lists: positional notation or named notation.

Positional notation

The default. Each value in the list of arguments supplied in the program call is associated with the parameter in the corresponding position.

Named notation

Explicitly associates the argument value with its parameter by name (not position). When you use named notation, you can supply the arguments in any order and you can omit IN arguments that have default values.

The call to the empid\_to\_name procedure is shown here with both notations:

BEGIN

-- Implicit positional notation.

empid\_to\_name(10, surname, first\_name);

-- Explicit named notation.

empid\_to\_name(in\_id=>10

,out\_last\_name=>surname

,out\_first\_name=>first\_name);

END;

You may combine positional and named notation, as long as positional arguments appear to the left of any named notation arguments; for example:

empid\_to\_name(10, surname, out\_first\_name => first\_name);

When calling stored functions from SQL, named notation is not supported.

### Local programs

A local program is a procedure or function that is defined in the declaration section of a PL/SQL block. The declaration of a local program must appear at the end of the declaration section, after the declarations of any types, records, cursors, variables, and exceptions. A program defined in a declaration section may only be referenced within that block's executable and exception sections. It is not defined outside that block.

The following program defines a local procedure and function:

PROCEDURE track\_revenue

IS

l\_total NUMBER;

PROCEDURE calc\_total (year\_in IN INTEGER) IS

BEGIN

calculations here ...

END;

FUNCTION below\_minimum (comp\_id IN INTEGER)

RETURN BOOLEAN

IS

BEGIN

...

END;

BEGIN

...main procedure logic here

END;

Local programs may be overloaded with the same restrictions as overloaded packaged programs.

### Program overloading

PL/SQL allows you to define two or more programs with the same name within any declaration section, including a package specification or body. This is called overloading. If two or more programs have the same name, they must be different in some other way so that the compiler can determine which program should be used.

Here is an example of overloaded programs in a built-in package specification:

PACKAGE DBMS\_OUTPUT

IS

PROCEDURE PUT\_LINE (a VARCHAR2);

PROCEDURE PUT\_LINE (a NUMBER);

PROCEDURE PUT\_LINE (a DATE);

END;

Each PUT\_LINE procedure is identical, except for the datatype of the parameter. That is enough difference for the compiler.

To overload programs successfully, one or more of the following conditions must be true:

* Parameters must differ by datatype family (number, character, datetime, or Boolean).
* The program type must be different (you can overload a function and a procedure of the same name and identical parameter list).
* The numbers of parameters must be different.

You cannot overload programs if:

* Only the datatypes of the functions' RETURN clauses are different.
* Parameter datatypes are within the same family (CHAR and VARCHAR2, NUMBER and INTEGER, etc.).
* Only the modes of the parameters are different.

### Forward declarations

Programs must be declared before they can be used. PL/SQL supports mutual recursion , in which program A calls program B, whereupon program B calls program A. To implement this mutual recursion, you must use a forward declaration of the programs. This technique declares a program in advance of the program definition, thus making it available for other programs to use. The forward declaration is the program header up to the IS/AS keyword:

PROCEDURE perform\_calc(year\_in IN NUMBER)

IS

/\* Forward declaration for total\_cost

function. \*/

FUNCTION total\_cost (...) RETURN NUMBER;

/\* The net\_profit function can now use

total\_cost. \*/

FUNCTION net\_profit(...) RETURN NUMBER

IS

BEGIN

RETURN total\_sales(...) - total\_cost(...);

END;

/\* The Total\_cost function calls net\_profit. \*/

FUNCTION total\_cost (...) RETURN NUMBER

IS

BEGIN

IF net\_profit(...) < 0

THEN

RETURN 0;

ELSE

RETURN...;

END IF;

END;

BEGIN /\* procedure perform\_calc \*/

...

END perform\_calc;

### Table functions

Table functions take a collection or REF CURSOR (set of rows) as input and return a collection of records (set of rows) as output. The PIPE ROW command is used to identify the input and output streams. This streamlined nature allows you to pipeline table functions together, eliminating the need to stage tables between transformations. Table functions typically appear in the FROM clause of SQL statements. For example:

CREATE FUNCTION pet\_family

(dad\_in IN pet\_t, mom\_in IN pet\_t)

RETURN pet\_nt PIPELINED IS

l\_count PLS\_INTEGER;

retval pet\_nt := pet\_nt ( );

BEGIN

PIPE ROW (dad\_in); -- identify streaming input

PIPE ROW (mom\_in); -- identify streaming input

IF mom\_in.breed = 'RABBIT' THEN l\_count := 12;

ELSIF mom\_in.breed = 'DOG' THEN l\_count := 4;

ELSIF mom\_in.breed = 'KANGAROO' THEN l\_count := 1;

END IF;

FOR indx IN 1 .. l\_count

LOOP

-- stream the results into the ouput pipeline

PIPE ROW (pet\_t ('BABY' || indx, mom\_in.breed

,SYSDATE));

END LOOP;

RETURN;

END;

### Compiling stored PL/SQL programs

The following keywords are available when creating Oracle9i stored programs:

*OR REPLACE*

Used to rebuild an existing program unit, preserving privileges granted on it.

*AUTHID*

Defines whether the program will execute with the privileges of, and resolve names like, the object owner (DEFINER), or as the user executing the function (CURRENT\_USER). Prior to Oracle8i, only the built-in packages DBMS\_SQL and DBMS\_UTILITY executed as CURRENT\_USER. The default AUTHID is DEFINER.

*DETERMINISTIC*

Required for function-based indexes. A function is DETERMINISTIC if it always returns the same value when called with the same parameters. Deterministic functions do not meaningfully reference package variables or the database. The built-in INITCAP is deterministic, but SYSDATE is not.

PARALLEL\_ENABLED [(PARTITION in\_parm BY {ANY HASH | RANGE}) ]

Tells the optimizer that a function is safe for parallel execution. The PARTITION BY clause is only available to functions that have a REF CURSOR IN parameter. This clause is used with table functions and tells the optimizer how the input can be partitioned.

*PIPELINED*

Used with table functions. Specifies that the results of this table function should be returned iteratively via the PIPE ROW command. A pipelined function can start to return data as it is generated instead of all at once after processing is complete.

*AGGREGATE USING*

Required for aggregate functions. Tells Oracle that the function evaluates a group of rows and returns a single result. For example, the built-in function AVG is an aggregate function.

### Native compilation of PL/SQL

With Oracle9i you can speed up many of your PL/SQL programs by compiling the stored programs natively. Oracle will translate your PL/SQL program into C code and compile it into a shared library (DLL on NT). You must have a supported C compiler on your database server machine to use native compilation. To compile natively, you must follow these steps:

* Edit the makefile, spnc\_makefile.mk, which you should find in the $ORACLE\_HOME/plsql subdirectory.
* Set the initialization parameter PLSQL\_COMPILER\_FLAGS = `NATIVE'. Individual developers may alter the value of PLSQL\_COMPILER\_FLAGS using the ALTER SESSION statement.
* The following parameters many also need to be set: PLSQL\_NATIVE\_C\_COMPILER, PLSQL\_NATIVE\_LINKER, PLSQL\_NATIVE\_LIBRARY\_DIR, PLSQL\_NATIVE\_MAKE\_UTILITY, and PLSQL\_NATIVE\_MAKE\_FILE\_NAME. The DBA can set these parameters in the Oracle initialization file or using an ALTER SYSTEM statement.
* Create or replace your stored programs.
* Verify the native compilation by querying the data dictionary view USER\_STORED\_SETTINGS and also by locating the shared library or DLL in the database server's file system.

### Privileges and stored PL/SQL

Stored SQL supports two models for addressing privileges at runtime. The default is definer rights, which tells Oracle that the privileges of the owner or definer of the program should be used. With the definer rights model, the owner of the program must have the required privileges granted directly to him—he cannot inherit the privileges from a role.

With invoker rights, the user who executes the program does so using his own privileges. Anonymous PL/SQL blocks always execute with invoker rights. To create a program that uses the invoker rights model, include the keywords AUTHID CURRENT\_USER in your program's declaration.

## Packages

A package is a collection of PL/SQL objects that are grouped together. There are a number of benefits to using packages, including information hiding, object-oriented design, top-down design, object persistence across transactions, and improved performance.

Elements that can be placed in a package include procedures, functions, constants, variables, cursors, exception names, and TYPE statements (for associative arrays [formerly known as index-by tables], records, REF CURSORs, etc.).

### Package Structure

A package can have two parts: the specification and the body. The package specification is required and lists all the objects that are publicly available (i.e., may be referenced from outside the package) for use in applications. It also provides all the information a developer needs in order to use objects in the package; essentially, it is the package's API.

The package body contains all the code needed to implement procedures, functions, and cursors listed in the specification, as well as any private objects (accessible only to other elements defined in that package), and an optional initialization section.

If a package specification does not contain any procedures or functions and no private code is needed, then that package does not need to have a package body.

The syntax for the package specification is:

CREATE [OR REPLACE] PACKAGE *package\_name*

[ AUTHID { CURRENT\_USER | DEFINER } ]

{ IS | AS }

[*definitions of public TYPEs*

*,declarations of public variables, types, and*

*objects*

*,declarations of exceptions*

*,pragmas*

*,declarations of cursors, procedures, and*

*functions*

*,headers of procedures and functions*]

END [*package\_name*];

The syntax for the package body is:

CREATE [OR REPLACE] PACKAGE BODY *package\_name*

{ IS | AS }

[*definitions of private TYPEs*

*,declarations of private variables, types, and*

*objects*

*,full definitions of cursors*

*,full definitions of procedures and functions*]

[BEGIN

*executable\_statements*

[EXCEPTION

*exception\_handlers* ] ]

END [*package\_name*];

The optional OR REPLACE keywords are used to rebuild an existing package, preserving any EXECUTE privileges previously granted to other accounts. The declarations in the specifications cannot be repeated in the body. Both the executable section and the exception section are optional in a package body. If the executable section is present, it is called the initialization section and it executes only once—the first time any package element is referenced during a session.

You must compile the package specification before the body specification. When you grant EXECUTE authority on a package to another schema or to PUBLIC, you are giving access only to the specification; the body remains hidden.

Here's an example of a package:

CREATE OR REPLACE PACKAGE time\_pkg IS

FUNCTION GetTimestamp RETURN DATE;

PRAGMA RESTRICT\_REFERENCES (GetTimestamp, WNDS);

PROCEDURE ResetTimestamp(new\_time DATE DEFAULT

SYSDATE);

END time\_pkg;

CREATE OR REPLACE PACKAGE BODY time\_pkg IS

StartTimeStamp DATE := SYSDATE;

-- StartTimeStamp is package data.

FUNCTION GetTimestamp RETURN DATE IS

BEGIN

RETURN StartTimeStamp;

END GetTimestamp;

PROCEDURE ResetTimestamp(new\_time DATE DEFAULT SYSDATE)

IS

BEGIN

StartTimeStamp := new\_time;

END ResetTimestamp;

END time\_pkg;

### Referencing Package Elements

The elements declared in the specification are referenced from the calling application via dot notation:

*package\_name.package\_element*

For example, the built-in package DBMS\_OUTPUT has a procedure PUT\_LINE, so a call to this package would look like this:

DBMS\_OUTPUT.PUT\_LINE('This is parameter data');

### Package Data

Data structures declared within a package specification or body, but outside any procedure or function in the package, are package data. The scope of package data is your entire session, spanning transaction boundaries and acting as globals for your programs.

Keep the following guidelines in mind as you work with package data:

* The state of your package variables is not affected by COMMITs and ROLLBACKs.
* A cursor declared in a package has global scope. It remains OPEN until you close it explicitly or until your session ends.
* A good practice is to hide your data structures in the package body and provide "get and set" programs to read and write that data. This technique can help protect your data.

### SERIALLY\_REUSABLE Pragma

If you need package data to exist only during a call to the packaged functions or procedures, and not between calls of the current session, you can potentially save runtime memory by using the pragma SERIALLY\_REUSABLE. After each call, PL/SQL closes the cursors and releases the memory used in the package. This technique is applicable only to large user communities executing the same routine. Normally, the database server's memory requirements grow linearly with the number of users; with SERIALLY\_REUSABLE, this growth can be less than linear, because work areas for package states are kept in a pool in the Oracle's System Global Area (SGA) and are shared among all users. This pragma must appear in both the specification and the body, as shown here:

CREATE OR REPLACE PACKAGE my\_pkg IS

PRAGMA SERIALLY\_REUSABLE;

PROCEDURE foo;

END my\_pkg;

CREATE OR REPLACE PACKAGE BODY my\_pkg IS

PRAGMA SERIALLY\_REUSABLE;

PROCEDURE foo IS

...

END my\_pkg;

### Package Initialization

The first time a user references a package element, the entire package is loaded into the SGA of the database instance to which the user is connected. That code is then shared by all sessions that have EXECUTE authority on the package.

Any package data are then instantiated into the session's User Global Area (UGA), a private area in either the System Global Area or the Program Global Area (PGA). If the package body contains an initialization section, that code will be executed. The initialization section is optional and appears at the end of the package body, beginning with a BEGIN statement and ending with the EXCEPTION section (if present) or the END of the package.

The following package initialization section runs a query to transfer the user's minimum balance into a global package variable. Programs can then reference the packaged variable (via the function) to retrieve the balance, rather than execute the query repeatedly:

CREATE OR REPLACE PACKAGE usrinfo

IS

FUNCTION minbal RETURN VARCHAR2;

END usrinfo;

/

CREATE OR REPLACE PACKAGE BODY usrinfo

IS

g\_minbal NUMBER; -- Package data

FUNCTION minbal RETURN VARCHAR2

IS BEGIN RETURN g\_minbal; END;

BEGIN -- Initialization section

SELECT minimum\_balance

INTO g\_minbal

FROM user\_configuration

WHERE username = USER;

EXCEPTION

WHEN NO\_DATA\_FOUND

THEN g\_minbal := NULL;

END usrinfo;

# Error checking - exception handling

## Exception Handling

PL/SQL allows developers to raise and handle errors (exceptions) in a very flexible and powerful way. Each PL/SQL block can have its own exception section in which exceptions can be trapped and handled (resolved or passed on to the enclosing block).

When an exception occurs (is raised) in a PL/SQL block, its execution section immediately terminates. Control is passed to the exception section.

Every exception in PL/SQL has an error number and error message; some exceptions also have names.

### Declaring Exceptions

Some exceptions (see the following table) have been pre-defined by Oracle in the STANDARD package or other built-in packages, such as UTL\_FILE. You can also declare your own exceptions as follows:

DECLARE

*exception\_name* EXCEPTION;

|  |  |
| --- | --- |
| **Error** | **Named exception** |
| ORA-00001 | DUP\_VAL\_ON\_INDEX |
| ORA-00051 | TIMEOUT\_ON\_RESOURCE |
| ORA-00061 | TRANSACTION\_BACKED\_OUT |
| ORA-01001 | INVALID\_CURSOR |
| ORA-01012 | NOT\_LOGGED\_ON |
| ORA-01017 | LOGIN\_DENIED |
| ORA-01403 | NO\_DATA\_FOUND |
| ORA-01410 | SYS\_INVALID\_ROWID |
| ORA-01422 | TOO\_MANY\_ROWS |
| ORA-01476 | ZERO\_DIVIDE |
| ORA-01725 | USERENV\_COMMMITSCN\_ERROR |
| ORA-01722 | INVALID\_NUMBER |
| ORA-06500 | STORAGE\_ERROR |
| ORA-06501 | PROGRAM\_ERROR |
| ORA-06502 | VALUE\_ERROR |
| ORA-06504 | ROWTYPE\_MISMATCH |
| ORA-06511 | CURSOR\_ALREADY\_OPEN |
| ORA-06530 | ACCESS\_INTO\_NULL |
| ORA-06531 | COLLECTION\_IS\_NULL |
| ORA-06532 | SUBSCRIPT\_OUTSIDE\_LIMIT |
| ORA-06533 | SUBSCRIPT\_BEYOND\_COUNT |
| ORA-09592 | CASE\_NOT\_FOUND |
| ORA-30625 | SELF\_IS\_NULL |
| ORA-29280 | INVALID\_PATH |
| ORA-29281 | INVALID\_MODE |
| ORA-29282 | INVALID\_FILEHANDLE |
| ORA-29283 | INVALID\_OPERATION |
| ORA-29284 | READ\_ERROR |
| ORA-29285 | WRITE\_ERROR |
| ORA-29286 | INTERNAL\_ERROR |
| ORA-29287 | INVALID\_MAXLINESIZE |
| ORA-29288 | INVALID\_FILENAME |
| ORA-29289 | ACCESS\_DENIED |
| ORA-29290 | INVALID\_OFFSET |
| ORA-29291 | DELETE\_FAILED |
| ORA-29292 | RENAME\_FAILED |

An exception can be declared only once in a block, but nested blocks can declare an exception with the same name as an outer block. If this multiple declaration occurs, scope takes precedence over name when handling the exception. The inner block's declaration takes precedence over a global declaration.

When you declare your own exception, you must RAISE it explicitly. All declared exceptions have an error code of 1 and the error message "User-defined exception," unless you use the EXCEPTION\_INIT pragma.

You can associate an error number with a declared exception with the PRAGMA EXCEPTION\_INIT statement using the following syntax:

DECLARE

*exception\_name* EXCEPTION;

PRAGMA EXCEPTION\_INIT (*exception\_name*,

*error\_number*);

where error\_number is a literal value (variable references are not allowed). This number can be an Oracle error, such as -1855, or an error in the user-definable -20000 to -20999 range.

### Raising Exceptions

An exception can be raised in three ways:

* By the PL/SQL runtime engine
* By an explicit RAISE statement in your code
* By a call to the built-in function RAISE\_APPLICATION\_ERROR

The syntax for the RAISE statement is:

RAISE *exception\_name*;

where exception\_name is the name of an exception that you have declared, or an exception that is declared in the STANDARD package. If you use the RAISE statement inside an exception handler, you can omit the exception name to re-raise the current exception:

RAISE;

This syntax is not valid outside the exception section.

The RAISE\_APPLICATION\_ERROR built-in function has the following header:

RAISE\_APPLICATION\_ERROR (

*num* BINARY\_INTEGER,

*msg* VARCHAR2,

*keeperrorstack* BOOLEAN DEFAULT FALSE);

where num is the error number (an integer between -20999 and -20000), msg is the associated error message, and keeperrorstack controls the contents of the error stack.

### Scope

The scope of an exception section is that portion of the code that is "covered" by the exception section. An exception handler will only handle or attempt to handle exceptions raised in the executable section of the PL/SQL block. Exceptions raised in the declaration or exception sections are automatically passed to the outer block. Any line or set of PL/SQL code can be placed inside its own block and given its own exception section. This allows you to limit the propagation of an exception.

### Propagation

Exceptions raised in a PL/SQL block propagate to an outer block if they are unhandled or re-raised in the exception section. When an exception occurs, PL/SQL looks for an exception handler that checks for the exception (or is the WHEN OTHERS clause) in the current block. If a match is not found, then PL/SQL propagates the exception to the enclosing block or calling program. This propagation continues until the exception is handled or propagated out of the outermost block, back to the calling program. In this case, the exception is "unhandled" and (1) stops the calling program, and (2) causes an automatic rollback of any outstanding transactions.

Once an exception is handled, it will not propagate upward. If you want to trap an exception, display a meaningful error message, and have the exception propagate upward as an error, you must re-raise the exception. The RAISE statement can re-raise the current exception or raise a new exception, as shown here:

PROCEDURE delete\_dept(deptno\_in IN NUMBER)

DECLARE

still\_have\_employees EXCEPTION

PRAGMA EXCEPTION\_INIT(still\_have\_employees.

-2292)

BEGIN

DELETE FROM dept

WHERE deptno = deptno\_in;

EXCEPTION

WHEN still\_have\_employees

THEN

DBMS\_OUTPUT.PUT\_LINE

('Please delete employees in dept first');

ROLLBACK;

RAISE; /\* Re-raise the current exception. \*/

END;

### WHEN OTHERS clause

Use the WHEN OTHERS clause in the exception handler as a catch-all to trap any exceptions that are not handled by specific WHEN clauses in the exception section. If present, this clause must be the last exception handler in the exception section. You specify this clause as follows:

EXCEPTION

WHEN OTHERS

THEN

...

### SQLCODE and SQLERRM

SQLCODE and SQLERRM are built-in functions that provide the SQL error code and message for the current exception. Use these functions inside the exception section's WHEN OTHERS clause to handle specific errors by number. The EXCEPTION\_INIT pragma allows you to handle errors by name. For example, the following code:

CREATE TABLE err\_test

(widget\_name VARCHAR2(100)

,widget\_count NUMBER

,CONSTRAINT no\_small\_numbers CHECK

(widget\_count > 1000));

BEGIN

INSERT INTO err\_test (widget\_name, widget\_count)

VALUES ('Athena',2);

EXCEPTION

WHEN OTHERS THEN

IF SQLCODE = -2290

AND SQLERRM LIKE '%NO\_SMALL\_NUMBERS%'

THEN

DBMS\_OUTPUT.PUT\_LINE('widget\_count is too

small');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Exception not handled,'

||'SQLcode='||SQLCODE);

DBMS\_OUTPUT.PUT\_LINE(SQLERRM);

END IF;

END;

produces this output:

widget\_count is too small

The built-in package DBMS\_UTILITY's FORMAT\_ERROR\_STACK and FORMAT\_CALL\_STACK procedures can be used to capture the full error stack and call stack. See the book Oracle Built-in Packages for more information on DBMS\_UTILITY.

### Exceptions and DML

When an exception is raised in a PL/SQL block, it does not roll back your current transaction, even if the block itself issued an INSERT, UPDATE, or DELETE. You must issue your own ROLLBACK statement if you want to clean up your transaction as a result of the exception.

If your exception goes unhandled (propagates out of the outermost block), however, most host environments will then force an automatic, unqualified rollback of any outstanding changes in your session.

# Triggers

Triggers are programs that execute in response to changes in table data or certain database events. There is a predefined set of events that can be "hooked" with a trigger, enabling you to integrate your own processing with that of the database. A triggering event fires or executes the trigger.

There are three types of triggering events:

* DML events fire when an INSERT, UPDATE, or DELETE statement executes.
* DDL events fire when a CREATE, ALTER, or DROP statement executes.
* Database events fire when one of the predefined database-level events occurs.

Complete lists of these events are included in later sections.

## Creating Triggers

The syntax for creating a trigger on a DML event is:

CREATE [OR REPLACE] TRIGGER *trigger\_name*

{ BEFORE | AFTER | INSTEAD OF } *trigger\_event*

ON {*table\_or\_view\_reference* |

NESTED TABLE *nested\_table\_column* OF *view*}

[REFERENCING [OLD AS *old*] [NEW AS *new*]

[PARENT AS *parent*]]

[FOR EACH ROW ][WHEN *trigger\_condition*]

*trigger\_body*;

The syntax for creating a trigger on a DDL or database event is:

CREATE [OR REPLACE] TRIGGER *trigger\_name*

{ BEFORE | AFTER } *trigger\_event*

ON [ DATABASE | *schema* ]

[WHEN *trigger\_condition*]

*trigger\_body*;

Trigger events are listed in the following table:

|  |  |
| --- | --- |
| **Trigger event** | **Description** |
| INSERT | Fires whenever a row is added to the table\_or\_view\_reference. |
| UPDATE | Fires whenever an UPDATE changes the table\_or\_view\_reference. UPDATE triggers can additionally specify an OF clause to restrict firing to updates OF certain columns. |
| DELETE | Fires whenever a row is deleted from the table\_or\_view\_reference. Does not fire on a TRUNCATE of the table. |
| ALTER | Fires whenever an ALTER statement changes a database object. In this context, objects are things like tables or packages (found in ALL\_OBJECTS). Can apply to a single schema or the entire database. |
| DROP | Fires whenever a DROP statement removes an object from the database. In this context, objects are things like tables or packages (found in ALL\_OBJECTS). Can apply to a single schema or the entire database. |
| SERVERERROR | Fires whenever a server error message is logged. Only AFTER triggers are allowed in this context. |
| LOGON | Fires whenever a session is created (a user connects to the database). Only AFTER triggers are allowed in this context. |
| LOGOFF | Fires whenever a session is terminated (a user disconnects from the database). Only BEFORE triggers are allowed in this context. |
| STARTUP | Fires when the database is opened. Only AFTER triggers are allowed in this context. |
| SHUTDOWN | Fires when the database is closed. Only BEFORE triggers are allowed in this context. |

Triggers can fire BEFORE or AFTER the triggering event. AFTER data triggers are slightly more efficient than BEFORE triggers.

The REFERENCING clause is allowed only for the data events INSERT, UPDATE, and DELETE. It lets you give a non-default name to the old and new pseudo-records. These pseudo-records give the program visibility to the pre- and post-change values in row-level triggers. These records are defined like %ROWTYPE records, except that columns of type LONG or LONG RAW cannot be referenced. They are prefixed with a colon in the trigger body, and referenced with dot notation. Unlike other records, these fields can only be assigned individually—aggregate assignment is not allowed. All old fields are NULL within INSERT triggers, and all new fields are NULL within DELETE triggers.

FOR EACH ROW defines the trigger to be a row-level trigger. Row-level triggers fire once for each row affected. The default is a statement-level trigger, which fires only once for each triggering statement.

The WHEN trigger\_condition specifies the conditions that must be met for the trigger to fire. Stored functions and object methods are not allowed in the trigger condition.

The trigger body is a standard PL/SQL block. For example:

CREATE OR REPLACE TRIGGER add\_tstamp

BEFORE INSERT ON emp

REFERENCING NEW as new\_row

FOR EACH ROW

BEGIN

-- Automatically timestamp the entry.

SELECT CURRENT\_TIMESTAMP

INTO :new\_row.entry\_timestamp

FROM dual;

END add\_tstamp;

Triggers are enabled on creation, and can be disabled (so they do not fire) with an ALTER statement, issued with the following syntax:

ALTER TRIGGER *trigger\_name* { ENABLE | DISABLE };

ALTER TABLE *table\_name* { ENABLE | DISABLE } ALL

TRIGGERS;

## Trigger Predicates

When using a single trigger for multiple events, use the trigger predicates INSERTING, UPDATING, and DELETING in the trigger condition to identify the triggering event, as shown in this example:

CREATE OR REPLACE TRIGGER emp\_log\_t

AFTER INSERT OR UPDATE OR DELETE ON emp

FOR EACH ROW

DECLARE

dmltype CHAR(1);

BEGIN

IF INSERTING THEN

dmltype := 'I';

INSERT INTO emp\_log (emp\_no, who, operation)

VALUES (:new.empno, USER, dmltype);

ELSIF UPDATING THEN

dmltype := 'U';

INSERT INTO emp\_log (emp\_no, who, operation)

VALUES (:new.empno, USER, dmltype);

END IF;

END;

## DML Events

The DML events include INSERT, UPDATE, and DELETE statements on a table or view. Triggers on these events can be statement-level triggers (table only) or row-level triggers and can fire BEFORE or AFTER the triggering event. BEFORE triggers can modify the data in affected rows, but perform an additional logical read. AFTER triggers do not perform this additional logical read, and therefore perform slightly better, but are not able to change the :new values. AFTER triggers are thus better suited for data validation functionality. Triggers cannot be created on SYS-owned objects. The order in which these triggers fire, if present, is as follows:

BEFORE statement-level trigger

For each row affected by the statement:

BEFORE row-level trigger

The triggering statement

AFTER row-level trigger

AFTER statement-level trigger

## DDL Events

The DDL events are CREATE, ALTER, and DROP. These triggers fire whenever the respective DDL statement is executed. DDL triggers can apply to either a single schema or the entire database.

## Database Events

The database events are SERVERERROR, LOGON, LOGOFF, STARTUP, and SHUTDOWN. Only BEFORE triggers are allowed for LOGOFF and SHUTDOWN events. Only AFTER triggers are allowed for LOGON, STARTUP, and SERVERERROR events. A SHUTDOWN trigger will fire on a SHUTDOWN NORMAL and a SHUTDOWN IMMEDIATE, but not on a SHUTDOWN ABORT.

# Cursors in PL/SQL

Every SQL statement executed by the RDBMS has a private SQL area that contains information about the SQL statement and the set of data returned. In PL/SQL, a cursor is a name assigned to a specific private SQL area for a specific SQL statement. There can be either static cursors, whose SQL statement is determined at compile time, or dynamic cursors, whose SQL statement is determined at runtime. Static cursors are used only for DML statements (SELECT, INSERT, UPDATE, DELETE, MERGE, or SELECT FOR UPDATE). These static cursors can be explicitly declared and named or may appear in-line as an implicit cursor. Dynamic cursors are used for any type of valid SQL statement including DDL (CREATE, TRUNCATE, ALTER) and DCL (GRANT, REVOKE). Dynamic cursors are implemented with the EXECUTE IMMEDIATE statement.

## Explicit Cursors

Explicit cursors are SELECT statements that are DECLAREd explicitly in the declaration section of the current block or in a package specification. Use OPEN, FETCH, and CLOSE in the execution or exception sections of your programs.

## Declaring explicit cursors

To use an explicit cursor, you must first declare it in the declaration section of a block or package. There are three types of explicit cursor declarations:

* A cursor without parameters; for example:
* CURSOR company\_cur
* IS

SELECT company\_id FROM company;

* A cursor that accepts arguments through a parameter list; for example:
* CURSOR company\_cur (id\_in IN NUMBER) IS
* SELECT name FROM company

WHERE company\_id = id\_in;

* A cursor header that contains a RETURN clause in place of the SELECT statement; for example:
* CURSOR company\_cur (id\_in IN NUMBER)

RETURN company%ROWTYPE;

This last example shows that the cursor can be declared separately from its implementation; for example, the header in a package specification and the implementation in the package body.

## Opening explicit cursors

To open a cursor, use the following syntax:

OPEN *cursor\_name* [(*argument* [,*argument* ...])];

where cursor\_name is the name of the cursor as declared in the declaration section. The arguments are required if the definition of the cursor contains a parameter list.

You must open an explicit cursor before you can fetch rows from that cursor. When the cursor is opened, the processing actually includes the parse, bind, open, and execute phases of SQL statement execution. This OPEN processing includes determining an execution plan, associating host variables and cursor parameters with the placeholders in the SQL statement, determining the result set, and, finally, setting the current row pointer to the first row in the result set.

When using a cursor FOR loop, the OPEN is implicit in the FOR statement. If you try to open a cursor that is already open, PL/SQL will raise an "ORA-06511: PL/SQL: cursor already open" exception.

## Fetching from explicit cursors

The FETCH statement places the contents of the current row into local variables. To retrieve all rows in a result set, each row needs to be fetched. The syntax for a FETCH statement is:

FETCH *cursor\_name* INTO *record\_or\_variable\_list*;

where cursor\_name is the name of the cursor as declared and opened.

## Closing explicit cursors

After all rows have been fetched, a cursor needs to be closed. Closing a cursor enables the PL/SQL memory optimization algorithm to release the associated memory at an appropriate time. You can close an explicit cursor by specifying a CLOSE statement as follows:

CLOSE *cursor\_name*;

where cursor\_name is the name of the cursor declared and opened.

If you declare a cursor in a local anonymous, procedure, or function block, that cursor will automatically close when the block terminates. Package-based cursors must be closed explicitly, or they stay open for the duration of your session. Closing a cursor that is not open raises an INVALID CURSOR exception.

## Explicit cursor attributes

There are four attributes associated with cursors: ISOPEN, FOUND, NOTFOUND, and ROWCOUNT. These attributes can be accessed with the % delimiter to obtain information about the state of the cursor. The syntax for a cursor attribute is:

*cursor\_name*%*attribute*

where cursor\_name is the name of the explicit cursor.

The behaviors of the explicit cursor attributes are described in the following table:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %ISOPEN | TRUE if cursor is open.  FALSE if cursor is not open. |
| %FOUND | INVALID\_CURSOR is raised if cursor has not been OPENed.  NULL before the first fetch.  TRUE if record was fetched successfully.  FALSE if no row was returned.  INVALID\_CURSOR if cursor has been CLOSEd. |
| %NOTFOUND | INVALID\_CURSOR is raised if cursor has not been OPENed.  NULL before the first fetch.  FALSE if record was fetched successfully.  TRUE if no row was returned.  INVALID\_CURSOR if cursor has been CLOSEd. |
| %ROWCOUNT | INVALID\_CURSOR is raised if cursor has not been OPENed.  The number of rows fetched from the cursor.  INVALID\_CURSOR if cursor has been CLOSEd. |

Frequently, a cursor attribute is checked as part of a WHILE loop that fetches rows from a cursor, as shown here:

DECLARE

caller\_rec caller\_pkg.caller\_cur%ROWTYPE;

BEGIN

OPEN caller\_pkg.caller\_cur;

LOOP

FETCH caller\_pkg.caller\_cur into caller\_rec;

EXIT WHEN caller\_pkg.caller\_cur%NOTFOUND

OR

caller\_pkg.caller\_cur%ROWCOUNT > 10;

UPDATE call

SET caller\_id = caller\_rec.caller\_id

WHERE call\_timestamp < SYSDATE;

END LOOP;

CLOSE caller\_pkg.caller\_cur;

END;

## Implicit Cursors

Whenever a SQL statement is directly in the execution or exception section of a PL/SQL block, you are working with implicit cursors. SQL statements handled this way include INSERT, UPDATE, DELETE, MERGE, and SELECT INTO. Unlike explicit cursors, implicit cursors do not need to be declared, OPENed, FETCHed, or CLOSEd.

SELECT statements handle the %FOUND and %NOTFOUND attributes differently from the way that explicit cursors do. When an implicit SELECT statement does not return any rows, PL/SQL immediately raises the NO\_DATA\_FOUND exception and control passes to the exception section. When an implicit SELECT returns more than one row, PL/SQL immediately raises the TOO\_MANY\_ROWS exception and control passes to the exception section.

Implicit cursor attributes are referenced via the SQL cursor. For example:

BEGIN

UPDATE activity SET last\_accessed := SYSDATE

WHERE UID = user\_id;

IF SQL%NOTFOUND THEN

INSERT INTO activity\_log (uid,last\_accessed)

VALUES (user\_id,SYSDATE);

END IF

END;

The following table lists the implicit cursor attributes:

|  |  |
| --- | --- |
| **Attributes** | **Description** |
| SQL%ISOPEN | Always FALSE because the cursor is opened implicitly and closed immediately after the statement is executed. |
| SQL%FOUND | NULL before the statement.  TRUE if one or more rows were inserted, merged, updated, or deleted or if only one row was selected.  FALSE if no row was selected, merged, updated, inserted, or deleted. |
| SQL%NOTFOUND | NULL before the statement.  TRUE if no row was selected, merged, updated, inserted, or deleted.  FALSE if one or more rows were inserted, merged, updated, or deleted. |
| SQL%ROWCOUNT | Number of rows affected by the cursor. |
| SQL%BULK\_ROWCOUNT | Pseudo index-by table containing the numbers of rows affected by the statements executed in bulk bind operations) |

Use the RETURNING clause in INSERT, UPDATE, and DELETE statements to obtain data modified by the associated DML statement. This clause allows you to avoid an additional SELECT statement to query the results of the DML statement. For example:

BEGIN

UPDATE activity SET last\_accessed := SYSDATE

WHERE UID = user\_id

RETURNING last\_accessed, cost\_center

INTO timestamp, chargeback\_acct;

## SELECT FOR UPDATE clause

By default, the Oracle RDBMS locks rows as they are changed. To lock all rows in a result set, use the FOR UPDATE clause in your SELECT statement when you OPEN the cursor, instead of when you change the data. Using the FOR UPDATE clause does not require you to actually make changes to the data; it only locks the rows when opening the cursor. These locks are released on the next COMMIT or ROLLBACK. As always, these row locks do not affect other SELECT statements unless they, too, are FOR UPDATE. The FOR UPDATE clause is appended to the end of the SELECT statement and has the following syntax:

SELECT ...

FROM ...

FOR UPDATE [OF *column\_reference*] [NOWAIT];

where column\_reference is a comma-delimited list of columns that appear in the SELECT clause. The NOWAIT keyword tells the RDBMS to not wait for other blocking locks to be released. The default is to wait forever.

In the following example, only columns from the inventory (pet) table are referenced FOR UPDATE, so no rows in the dog\_breeds (dog) table are locked when hounds\_in\_stock\_cur is opened:

DECLARE

CURSOR hounds\_in\_stock\_cur IS

SELECT pet.stock\_no, pet.breeder, dog.size

FROM dog\_breeds dog ,inventory pet

WHERE dog.breed = pet.breed

AND dog.class = 'HOUND'

FOR UPDATE OF pet.stock\_no, pet.breeder;

BEGIN

## WHERE CURRENT OF clause

UPDATE and DELETE statements can use a WHERE CURRENT OF clause if they reference a cursor declared FOR UPDATE. This syntax indicates that the UPDATE or DELETE should modify the current row identified by the FOR UPDATE cursor. The syntax is:

[UPDATE | DELETE ] ...

WHERE CURRENT OF *cursor\_name*;

By using WHERE CURRENT OF, you do not have to repeat the WHERE clause in the SELECT statement. For example:

DECLARE

CURSOR wip\_cur IS

SELECT acct\_no, enter\_date FROM wip

WHERE enter\_date < SYSDATE - 7

FOR UPDATE;

BEGIN

FOR wip\_rec IN wip\_cur

LOOP

INSERT INTO acct\_log (acct\_no, order\_date)

VALUES (wip\_rec.acct\_no, wip\_rec.enter\_

date);

DELETE FROM wip

WHERE CURRENT OF wip\_cur;

END LOOP;

END;

## Dynamic Cursors

Dynamic cursors are implemented with an EXECUTE IMMEDIATE statement together with the OPEN FOR, FETCH, and CLOSE statements. The EXECUTE IMMEDIATE statement supports single-row queries and DDL, while the OPEN FOR, FETCH, and CLOSE statements support dynamic multi-row queries. The syntax for these statements is:

EXECUTE IMMEDIATE *sql\_statement*

[INTO {*variable* [,*variable* ...] | *record*}]

[USING [IN | OUT | IN OUT] *bind\_argument*

[,[IN | OUT | IN OUT] *bind\_argument* ...] ]

[{RETURNING | RETURN} INTO *bind\_argument* [,*bind\_argument*]...];

The EXECUTE IMMEDIATE statement parses and executes the SQL statement in a single step. The EXECUTE IMMEDIATE statement requires a terminating semicolon, but the sql\_statement must not have a trailing semicolon. For example:

EXECUTE IMMEDIATE 'TRUNCATE TABLE foo';

EXECUTE IMMEDIATE 'GRANT SELECT ON '|| tabname\_v ||

' TO ' || grantee\_list;

The OPEN FOR statement assigns a multi-row query to a weakly typed cursor variable. The rows are then FETCHed and the cursor CLOSEd:

DECLARE

TYPE cv\_typ IS REF CURSOR;

cv cv\_typ;

laccount\_no NUMBER;

lbalance NUMBER;

BEGIN

OPEN cv FOR

'SELECT account\_no, balance

FROM accounts

WHERE balance < 500';

LOOP

FETCH cv INTO laccount\_no, lbalance;

EXIT WHEN cv%NOTFOUND;

-- Process the row.

END LOOP;

CLOSE cv;

END;

Because SQL statements usually execute repeatedly, declare your dynamic cursor with bind variables and pass the values to Oracle at runtime. The parsed form of the statement can be reused from the shared pool, improving performance. For example:

EXECUTE IMMEDIATE 'INSERT INTO hr.regions

(region\_id, region\_name) VALUES (:r\_id, :r\_name)'

USING id, name;

## Cursor Variables

A cursor variable is a data structure that points to a cursor object, which in turn points to the cursor's result set. You can use cursor variables to more easily retrieve rows in a result set from client and server programs. You can also use cursor variables to hide minor variations in queries.

The syntax for a REF\_CURSOR type (cursor variable) is:

TYPE *ref\_cursor\_name* IS REF CURSOR

[RETURN *record\_type*];

If you do not include a RETURN clause, then you are declaring a weak REF CURSOR. Cursor variables declared from weak REF CURSORs can be associated with any query at runtime. A REF CURSOR declaration with a RETURN clause defines a "strong" REF CURSOR. A cursor variable based on a strong REF CURSOR can be associated with queries whose result sets match the number and datatype of the record structure after the RETURN at runtime.

To use cursor variables, you must first create a REF\_CURSOR type, then declare a cursor variable based on that type.

The following example shows the use of both weak and strong REF CURSORs:

DECLARE

-- Create a cursor type based on the company's

table.

TYPE company\_curtype IS REF CURSOR

RETURN companies%ROWTYPE;

-- Create the variable based on the REF CURSOR.

company\_cur company\_curtype;

-- And now the weak, general approach.

TYPE any\_curtype IS REF CURSOR;

generic\_curvar any\_curtype;

The syntax to OPEN a cursor variable is:

OPEN *cursor\_name* FOR *SELECT\_statement*;

FETCH and CLOSE a cursor variable using the same syntax as for explicit cursors. There are a number of restrictions on cursor variables:

* You cannot declare package-level cursor variables because they do not have a persistent state. (You can declare them in packaged procedures and functions, however.)
* You cannot assign NULLs to a cursor variable nor can you use comparison operators to test for equality, inequality, or nullity.
* Neither database columns nor collections can store cursor variables.
* You cannot use RPCs to pass cursor variables from one server to another.

## Cursor Expressions

A cursor expression is a cursor that is used as a column expression in the SELECT list of an explicit cursor. The syntax for a cursor expression is:

CURSOR (*subquery*)

Cursor expressions can reduce the amount of redundant data returned to a calling program over techniques that involve joining the tables together. The cursor expression is automatically opened when the parent row is fetched. Cursor expressions can be nested as well. These nested cursors are closed when one of the following occurs:

* The nested cursor is explicitly closed by the program.
* The parent cursor is closed.
* The parent cursor is re-executed.
* An exception is raised during the fetch of the parent row.

An example of a cursor expression follows:

DECLARE

TYPE refcursor IS REF CURSOR;

CURSOR order\_cur IS

SELECT o.order\_date ,o.order\_status

,CURSOR(SELECT p.translated\_name

,i.unit\_price

,i.quantity

FROM oe.order\_items i

,oe.product\_descriptions p

WHERE i.product\_id = p.product\_id

AND i.order\_id = o.order\_id)

FROM oe.orders o

WHERE order\_date BETWEEN TO\_DATE('01-Jan-03')

AND TO\_DATE('31-Jan\_03');

odate oe.orders.order\_date%TYPE;

ostatus oe.orders.order\_status%TYPE;

od\_cur refcursor;

tname oe.product\_descriptions.translated\_name%TYPE;

price oe.order\_items.unit\_price%TYPE;

qty oe.order\_items.quantity%TYPE;

BEGIN

OPEN order\_cur;

LOOP

FETCH order\_cur INTO odate, ostatus, od\_cur;

EXIT WHEN order\_cur%NOTFOUND;

LOOP

FETCH od\_cur INTO tname, price, qty;

EXIT WHEN od\_cur%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(odate||','||ostatus

||','||tname||','||price||','||qty);

END LOOP;

END LOOP;

CLOSE order\_cur;

END;