14

# PL/SQL Language Elements

Summarizes the syntax and semantics of PL/SQL language elements and provides links to examples and related topics.

For instructions for reading the syntax diagrams, see *Oracle Database SQL Language Reference*.

**Topics**


- ACCESSIBLE BY Clause
- AGGREGATE Clause
- Assignment Statement
- AUTONOMOUS_TRANSACTION Pragma
- Basic LOOP Statement
- Block
- Call Specification
- CASE Statement
- CLOSE Statement
- Collection Method Invocation
- Collection Variable Declaration
- Comment
- COMPILE Clause
- Constant Declaration
- CONTINUE Statement
- COVERAGE Pragma
- Cursor FOR LOOP Statement
- Cursor Variable Declaration
- Datatype Attribute
- DEFAULT COLLATION Clause
- DELETE Statement Extension
- DEPRECATE Pragma
- DETERMINISTIC Clause
- Element Specification
- EXCEPTION_INIT Pragma
- Exception Declaration
- Exception Handler
- EXECUTE IMMEDIATE Statement

- EXIT Statement
- Explicit Cursor Declaration and Definition
- Expression
- FETCH Statement
- FOR LOOP Statement
- FORALL Statement
- Formal Parameter Declaration
- Function Declaration and Definition
- GOTO Statement
- IF Statement
- Implicit Cursor Attribute
- INLINE Pragma
- Invoker's Rights and Definer's Rights Clause
- INSERT Statement Extension
- Iterator
- Named Cursor Attribute
- NULL Statement
- OPEN Statement
- OPEN FOR Statement
- PARALLEL_ENABLE Clause
- PIPE ROW Statement
- PIPELINED Clause
- Procedure Declaration and Definition
- Qualified Expression
- RAISE Statement
- Record Variable Declaration
- RESTRICT_REFERENCES Pragma (deprecated)
- RETURN Statement
- RETURNING INTO Clause
- RESULT_CACHE Clause
- %ROWTYPE Attribute
- Scalar Variable Declaration
- SELECT INTO Statement
- SERIALLY_REUSABLE Pragma
- SHARING Clause
- SQL_MACRO Clause
- SQLCODE Function
- SQLERRM Function

- SUPPRESSES_WARNING_6009 Pragma
- %TYPE Attribute
- UDF Pragma
- UPDATE Statement Extensions
- WHILE LOOP Statement

> **See Also:**
>
> - PL/SQL Language Fundamentals

# ACCESSIBLE BY Clause

The `ACCESSIBLE BY` clause restricts access to a unit or subprogram by other units.

The **accessor list** explicitly lists those units which may have access. The accessor list can be defined on individual subprograms in a package. This list is checked in addition to the accessor list defined on the package itself (if any). This list may only restrict access to the subprogram – it cannot expand access. This code management feature is useful to prevent inadvertent use of internal subprograms. For example, it may not be convenient or feasible to reorganize a package into two packages: one for a small number of procedures requiring restricted access, and another one for the remaining units requiring public access.

The `ACCESSIBLE BY` clause may appear in the declarations of object types, object type bodies, packages, and subprograms.

The `ACCESSIBLE BY` clause can appear in the following SQL statements:

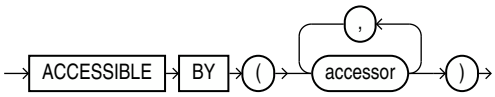
- ALTER TYPE Statement
- CREATE FUNCTION Statement
- CREATE PROCEDURE Statement
- CREATE PACKAGE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

**Topics**

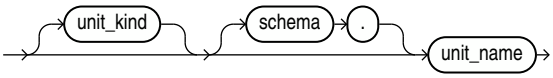
- Syntax
- Semantics
- Usage Notes
- Examples
- Related Topics

### Syntax

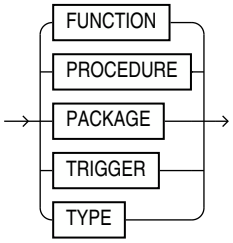
***accessible_by_clause* ::=**




***accessor* ::=**




***unit_kind* ::=**




### Semantics

***accessible_by_clause***

***accessor***

**[*schema*.]*unit_name***

Specifies a stored PL/SQL unit that can invoke the entity.

Each `accessor` specifies another PL/SQL entity that may access the entity which includes the `ACCESSIBLE BY` clause.

When an `ACCESSIBLE BY` clause appears, only entities named in the clause may access the entity in which the clause appears.

An `accessor` may appear more than once in the `ACCESSIBLE BY` clause.

The `ACCESSIBLE BY` clause can appear only once in the unit declaration.

An entity named in an `accessor` is not required to exist.

When an entity with an `ACCESSIBLE BY` clause is invoked, it imposes an additional access check after all other checks have been performed. These checks are:

- The invoked unit must include an `accessor` with the same *unit_name* and *unit_kind* as the invoking unit.

- If the `accessor` includes a *schema*, the invoking unit must be in that *schema*.
- If the `accessor` does not include a *schema*, the invoker must be from the same schema as the invoked entity.

***unit_kind***

Specifies if the unit is a `FUNCTION`, `PACKAGE`, `PROCEDURE`, `TRIGGER`, or `TYPE`.

### Usage Notes

The *unit_kind* is optional, but it is recommended to specify it to avoid ambiguity when units have the same name. For example, it is possible to define a trigger with the same name as a function.

The `ACCESSIBLE BY` clause allows access only when the call is direct. The check will fail if the access is through static SQL, `DBMS_SQL`, or dynamic SQL.

Any call to the initialization procedure of a package specification or package body will be checked against the *accessor* list of the package specification.

A unit can always access itself. An item in a unit can reference another item in the same unit.

RPC calls to a protected subprogram will always fail, since there is no context available to check the validity of the call, at either compile-time or run-time.

Calls to a protected subprogram from a conditional compilation directive will fail.

### Examples

#### Example 14-1 Restricting Access to Top-Level Procedures in the Same Schema

This example shows that the top-level procedure top_protected_proc can only be called by procedure top_trusted_proc in the current schema. The user cannot call top_proctected_proc directly.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Restricting Access to Top-Level Procedures in the Same Schema

```
PROCEDURE top_protected_proc
  ACCESSIBLE BY (PROCEDURE top_trusted_proc)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Processed top_protected_proc.');
END;

PROCEDURE top_trusted_proc AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls top_protected_proc');
  top_protected_proc;
END;


EXEC top_trusted_proc;
```

```
top_trusted_proc calls top_protected_proc
Processed top_protected_proc.


EXEC top_protected_proc;
BEGIN top_protected_proc; END;

PLS-00904: insufficient privilege to access object TOP_PROTECTED_PROC
```

### Example 14-2 Restricting Access to a Unit Name of Any Kind

This example shows that if the PL/SQL *unit_kind* is not specified in the `ACCESSIBLE BY` clause, then a call from any unit kind is allowed if the unit name matches. There is no compilation error if the *unit_kind* specified in the `ACCESSIBLE BY` clause does not match any existing objects. It is possible to define a trigger with the same name as a function. It is recommended to specify the *unit_kind*.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Restricting Access to a Unit Name of Any Kind

```
PROCEDURE protected_proc2
  ACCESSIBLE BY (top_trusted_f)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Processed protected_proc2.');
END;

FUNCTION top_protected_f RETURN NUMBER
ACCESSIBLE BY (TRIGGER top_trusted_f ) AS
BEGIN
   RETURN 0.5;
END top_protected_f;

FUNCTION top_trusted_f RETURN NUMBER AUTHID DEFINER IS
  FUNCTION g RETURN NUMBER DETERMINISTIC IS
  BEGIN
     RETURN 0.5;
  END g;
BEGIN
  protected_proc2;
  RETURN g() - DBMS_RANDOM.VALUE();
END top_trusted_f;

SELECT top_trusted_f FROM DUAL;
   .381773176

1 row selected.

Processed protected_proc2.
```

### Example 14-3 Restricting Access to a Stored Procedure

This example shows a package procedure that can only be called by top_trusted_proc procedure. The `ACCESSIBLE BY` clause of a subprogram specification and body must match. A compilation error is raised if a call is made to an existing procedure with an `ACCESSIBLE BY` clause that does not include this procedure in its accessor list.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Restricting Access to a Stored Procedure

```
CREATE OR REPLACE PACKAGE protected_pkg
AS
  PROCEDURE public_proc;
  PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc);
END;

CREATE OR REPLACE PACKAGE BODY protected_pkg
AS
  PROCEDURE public_proc AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Processed protected_pkg.public_proc');
  END;
  PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc) AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Processed protected_pkg.private_proc');
  END;
END;

CREATE OR REPLACE PROCEDURE top_trusted_proc
AS
  BEGIN
     DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls protected_pkg.private_proc
');
     protected_pkg.private_proc;
  END;

Procedure created.

EXEC top_trusted_proc;
top_trusted_proc calls protected_pkg.private_proc
Processed protected_pkg.private_proc

EXEC protected_pkg.private_proc
PLS-00904: insufficient privilege to access object PRIVATE_PROC
```

### Related Topics

In this chapter:

- Function Declaration and Definition
- Procedure Declaration and Definition

In other chapters:

- Nested, Package, and Standalone Subprograms
- Subprogram Properties
- Package Writing Guidelines

# AGGREGATE Clause

Identifies the function as an **aggregate function**, or one that evaluates a group of rows and returns a single row.

You can specify aggregate functions in the select list, `HAVING` clause, and `ORDER BY` clause.

When you specify a user-defined aggregate function in a query, you can treat it as an **analytic function** (one that operates on a query result set). To do so, use the `OVER` *`analytic_clause`* syntax available for SQL analytic functions.

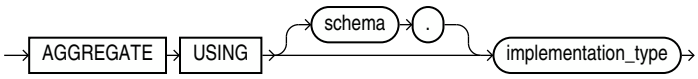
The `AGGREGATE` clause can appear in the CREATE FUNCTION Statement.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***aggregate_clause* ::=**




### Semantics

***aggregate_clause***

**AGGREGATE USING**

Specify the name of the implementation type of the function.

***[schema.] implementation_type***

The implementation type must be an ADT containing the implementation of the `ODCIAggregate` subprograms. If you do not specify *`schema`*, then the database assumes that the implementation type is in your schema.

**Restriction on AGGREGATE USING**

You cannot specify the *aggregate_clause* for a nested function.

If you specify this clause, then you can specify only one input argument for the function.

### Examples

- Example 13-34, "Pipelined Table Function as Aggregate Function"

### Related Topics

In this chapter:

- Function Declaration and Definition

In other books:

- *Oracle Database SQL Language Reference* for syntax and semantics of analytic functions
- *Oracle Database Data Cartridge Developer's Guide* for more information about user-defined aggregate functions
- *Oracle Database Data Cartridge Developer's Guide* for information about ODCI subprograms

# Assignment Statement

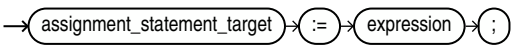
The assignment statement sets the value of a data item to a valid value.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

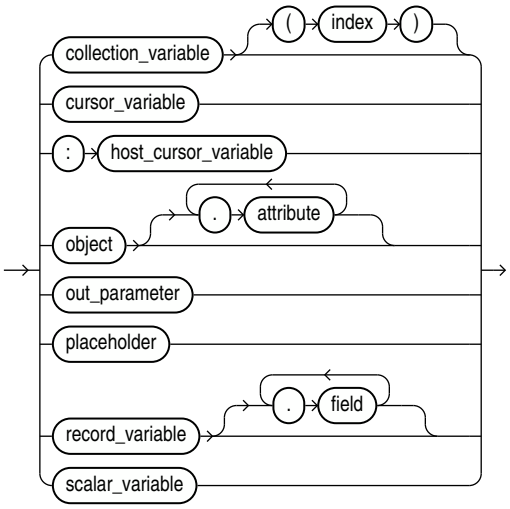
### Syntax

***assignment_statement* ::=**

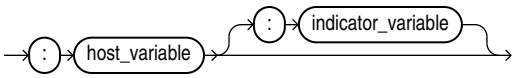



(*expression* ::=)

***assignment_statement_target* ::=**




***placeholder* ::=**




### Semantics

***assignment_statement***

***expression***

Expression whose value is to be assigned to *assignment_statement_target*.

*expression* and *assignment_statement_target* must have compatible data types.

> **Note:**
>
> Collections with elements of the same type might not have the same data type. For the syntax of collection type definitions, see "Collection Variable Declaration".

***assignment_statement_target***

Data item to which the value of *expression* is to be assigned.

***collection_variable***

Name of a collection variable.

***index***

Index of an element of *collection_variable*. Without *index*, the entire collection variable is the assignment statement target.

*index* must be a numeric expression whose data type either is `PLS_INTEGER` or can be implicitly converted to `PLS_INTEGER` (for information about the latter, see "Predefined PLS_INTEGER Subtypes").

***cursor_variable***

Name of a cursor variable.

**:*host_cursor_variable***

Name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *host_cursor_variable*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

***object***

Name of an instance of an abstract data type (ADT).

***attribute***

Name of an attribute of *object*. Without *attribute*, the entire ADT is the assignment statement target.

***out_parameter***

Name of a formal `OUT` or `IN OUT` parameter of the subprogram in which the assignment statement appears.

***record_variable***

Name of a record variable.

***field***

Name of a field of *record_variable*. Without *field*, the entire record variable is the assignment statement target.

***scalar_variable***

Name of a PL/SQL scalar variable.

***placeholder***

**:*host_variable***

Name of a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *host_variable*.

**:*indicator_variable***

Name of an indicator variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. (An indicator variable indicates the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, an indicator variable can a detect null or truncated value in an output host variable.) Do not put space between *host_variable* and the colon (:) or between the colon and *indicator_variable*. This is correct:

```
:host_variable:indicator_variable
```

### Examples

- Example 3-24, "Assigning Values to Variables with Assignment Statement"
- Example 3-27, "Assigning Value to BOOLEAN Variable"
- Example 6-14, "Data Type Compatibility for Collection Assignment"

### Related Topics

In this chapter:

- "Expression"
- "FETCH Statement"
- "SELECT INTO Statement"

In other chapters:

- "Assigning Values to Variables"
- "Assigning Values to Collection Variables"
- "Assigning Values to Record Variables"

# AUTONOMOUS_TRANSACTION Pragma

The `AUTONOMOUS_TRANSACTION` pragma marks a routine as **autonomous**; that is, independent of the main transaction.

In this context, a **routine** is one of these:

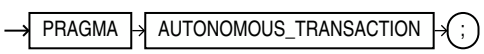
- Schema-level (not nested) anonymous PL/SQL block
- Standalone, package, or nested subprogram
- Method of an ADT
- Noncompound trigger

### Topics

- Syntax
- Examples
- Related Topics

### Syntax

***autonomous_trans_pragma* ::=**




### Examples

- Example 7-43, "Declaring Autonomous Function in Package"
- Example 7-44, "Declaring Autonomous Standalone Procedure"

- Example 7-45, "Declaring Autonomous PL/SQL Block"
- Example 7-46, "Autonomous Trigger Logs INSERT Statements"
- Example 7-47, "Autonomous Trigger Uses Native Dynamic SQL for DDL"
- Example 7-48, "Invoking Autonomous Function"

**Related Topics**

- Pragmas
- Autonomous Transactions

# Basic LOOP Statement

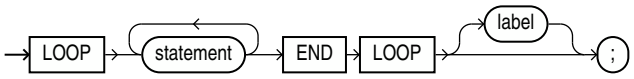
With each iteration of the basic `LOOP` statement, its statements run and control returns to the top of the loop. The `LOOP` statement ends when a statement inside the loop transfers control outside the loop or raises an exception.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***basic_loop_statement* ::=**




(*statement* ::=)

**Semantics**

***basic_loop_statement***

***statement***

To prevent an infinite loop, at least one statement must transfer control outside the loop. The statements that can transfer control outside the loop are:

- "CONTINUE Statement" (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement"
- "GOTO Statement"
- "RAISE Statement"

***label***

A label that identifies *basic_loop_statement* (see "statement ::=" and "label"). `CONTINUE`, `EXIT`, and `GOTO` statements can reference this label.

Labels improve readability, especially when `LOOP` statements are nested, but only if you ensure that the label in the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

### Examples

#### Example 14-4 Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements

In this example, one basic `LOOP` statement is nested inside the other, and both have labels. The inner loop has two `EXIT WHEN` statements; one that exits the inner loop and one that exits the outer loop.

```
DECLARE
  s  PLS_INTEGER := 0;
  i  PLS_INTEGER := 0;
  j  PLS_INTEGER;
BEGIN
  <<outer_loop>>
  LOOP
    i := i + 1;
    j := 0;
    <<inner_loop>>
    LOOP
      j := j + 1;
      s := s + i * j; -- Sum several products
      EXIT inner_loop WHEN (j > 5);
      EXIT outer_loop WHEN ((i * j) > 15);
    END LOOP inner_loop;
  END LOOP outer_loop;
  DBMS_OUTPUT.PUT_LINE
    ('The sum of products equals: ' || TO_CHAR(s));
END;
/
```

Result:

```
The sum of products equals: 166
```

#### Example 14-5 Nested, Unabeled Basic LOOP Statements with EXIT WHEN Statements

An `EXIT WHEN` statement in an inner loop can transfer control to an outer loop only if the outer loop is labeled.

In this example, the outer loop is not labeled; therefore, the inner loop cannot transfer control to it.

```
DECLARE
  i PLS_INTEGER := 0;
  j PLS_INTEGER := 0;

BEGIN
  LOOP
    i := i + 1;
    DBMS_OUTPUT.PUT_LINE ('i = ' || i);

    LOOP
      j := j + 1;
      DBMS_OUTPUT.PUT_LINE ('j = ' || j);
      EXIT WHEN (j > 3);
    END LOOP;
```

```
    DBMS_OUTPUT.PUT_LINE ('Exited inner loop');

    EXIT WHEN (i > 2);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Exited outer loop');
END;
/
```

Result:

```
i = 1
j = 1
j = 2
j = 3
j = 4
Exited inner loop
i = 2
j = 5
Exited inner loop
i = 3
j = 6
Exited inner loop
Exited outer loop

PL/SQL procedure successfully completed.
```

### Related Topics

- "Cursor FOR LOOP Statement"
- "FOR LOOP Statement"
- "WHILE LOOP Statement"
- "Basic LOOP Statement"

# Block

The **block**, which groups related declarations and statements, is the basic unit of a PL/SQL source program.

It has an optional declarative part, a required executable part, and an optional exception-handling part. Declarations are local to the block and cease to exist when the block completes execution. Blocks can be nested.

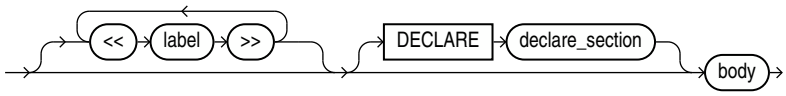
An anonymous block is an executable statement.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

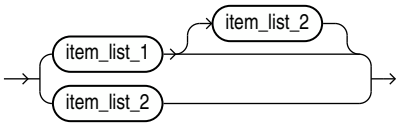
### Syntax

***plsql_block* ::=**



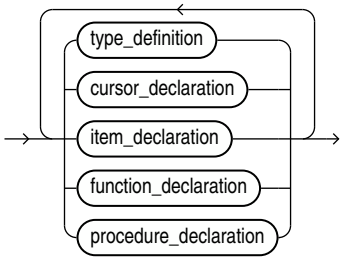

(*body* ::=)

***declare_section* ::=**



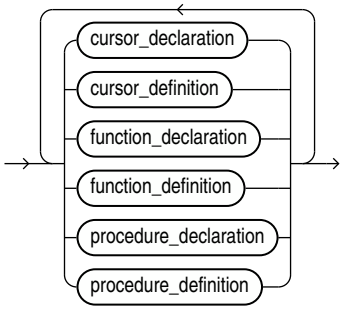

(*item_list_2* ::=)

***item_list_1* ::=**



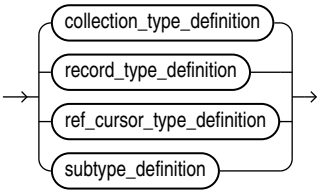

(*cursor_declaration* ::=, *function_declaration* ::=, *item_declaration* ::=, *procedure_declaration* ::=, *type_definition* ::=)

***item_list_2* ::=**



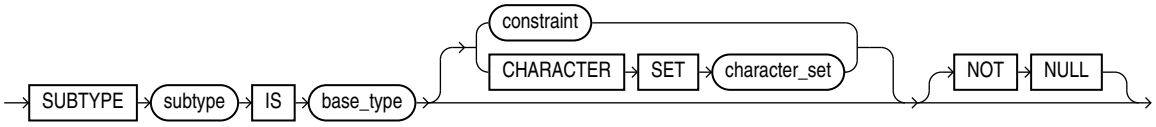
(*cursor_declaration* ::=, *cursor_definition* ::=, *function_declaration* ::=, *function_definition* ::=, *procedure_declaration* ::=, *procedure_definition* ::=)

***type_definition* ::=**

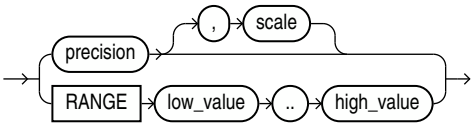



(*collection_type_definition* ::=, *record_type_definition* ::=, *ref_cursor_type_definition* ::=, *subtype_definition* ::=)

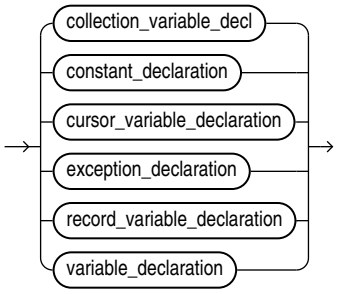
***subtype_definition* ::=**




***constraint* ::=**

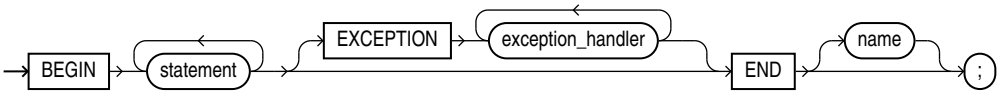



***item_declaration* ::=**



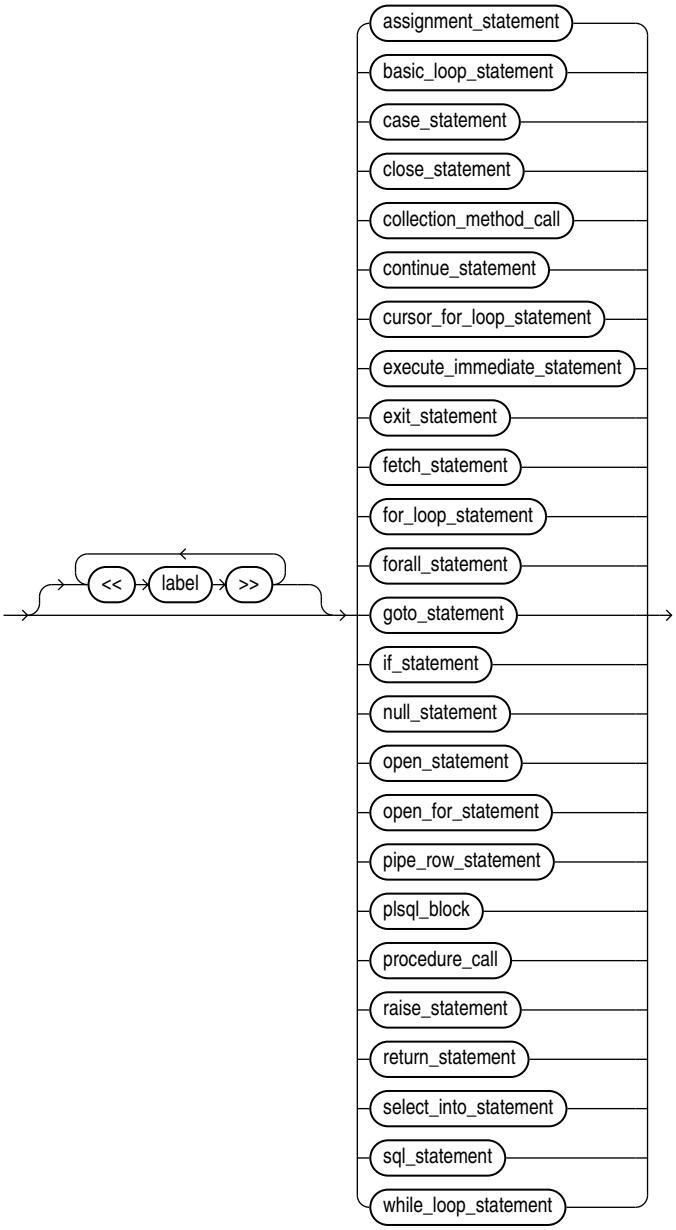

(*collection_variable_decl* ::=, *constant_declaration* ::=, *cursor_declaration* ::=, *cursor_variable_declaration* ::=, *exception_declaration* ::=, *record_variable_declaration* ::=, *variable_declaration* ::=)

***body* ::=**



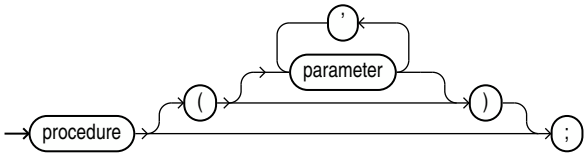

(*exception_handler* ::=)

***statement* ::=**

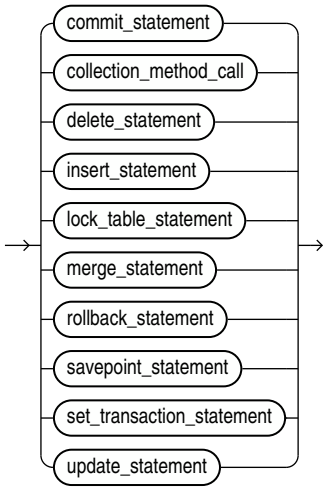



(*plsql_block* ::=, *procedure_call* ::=, *sql_statement* ::=)

***procedure_call* ::=**




***sql_statement* ::=**




### Semantics

#### *plsql_block*

#### *label*

Undeclared identifier, unique for the block.

#### DECLARE

Starts the declarative part of the block.

#### *declare_section*

Contains local declarations, which exist only in the block and its sub-blocks and are not visible to enclosing blocks.

#### Restrictions on *declare_section*

- A *declare_section* in *create_package*, *create_package_body*, or *compound_trigger_block* cannot include PRAGMA AUTONOMOUS_TRANSACTION.
- A *declare_section* in *trigger_body* or *tps_body* cannot declare variables of the data type LONG or LONG RAW.

> **See Also:**
>
> - "CREATE PACKAGE Statement" for more information about *create_package*
> - "CREATE PACKAGE BODY Statement" for more information about *create_package_body*
> - "CREATE TRIGGER Statement" for more information about *compound_trigger_block*, *trigger_body*, and *tps_body*

***subtype_definition***

Static expressions can be used in subtype declarations. See Static Expressions for more information.

***subtype***

Name of the user-defined subtype that you are defining.

***base_type***

Base type of the subtype that you are defining. *base_type* can be any scalar or user-defined PL/SQL datatype specifier such as `CHAR`, `DATE`, or `RECORD`.

**CHARACTER SET *character_set***

Specifies the character set for a subtype of a character data type.

Restriction on CHARACTER SET *character_set*

Do not specify this clause if *base_type* is not a character data type.

**NOT NULL**

Imposes the `NOT NULL` constraint on data items declared with this subtype. For information about this constraint, see "NOT NULL Constraint".

***constraint***

Specifies a constraint for a subtype of a numeric data type.

Restriction on *constraint*

Do not specify *constraint* if *base_type* is not a numeric data type.

***precision***

Specifies the precision for a constrained subtype of a numeric data type.

Restriction on *precision*

Do not specify *precision* if *base_type* cannot specify precision.

***scale***

Specifies the scale for a constrained subtype of a numeric data type.

Restriction on *scale*

Do not specify *scale* if *base_type* cannot specify scale.

**RANGE *low_value .. high_value***

Specifies the range for a constrained subtype of a numeric data type. The *low_value* and *high_value* must be numeric literals.

**Restriction on RANGE *high_value .. low_value***

Specify this clause only if *base_type* is PLS_INTEGER or a subtype of PLS_INTEGER (either predefined or user-defined). (For a summary of the predefined subtypes of PLS_INTEGER, see Table 4-3. For information about user-defined subtypes with ranges, see "Constrained Subtypes".)

***body***

**BEGIN**

Starts the executable part of the block, which contains executable statements.

**EXCEPTION**

Starts the exception-handling part of the block. When PL/SQL raises an exception, normal execution of the block stops and control transfers to the appropriate *exception_handler*. After the exception handler completes, execution resumes with the statement following the block. For more information about exception-handling, see PL/SQL Error Handling.

***exception_handler***

See "Exception Handler".

**END**

Ends the block.

***name***

The name of the block to which END applies—a label, function name, procedure name, or package name.

***statement***

***label***

Undeclared identifier, unique for the statement.

***assignment_statement***

See "Assignment Statement".

***basic_loop_statement***

See "Basic LOOP Statement".

***case_statement***

See "CASE Statement".

***close_statement***

See "CLOSE Statement".

***collection_method_call***

Invocation of one of these collection methods, which are procedures:

- `DELETE`
- `EXTEND`
- `TRIM`

For syntax, see "Collection Method Invocation".

***continue_statement***

See "CONTINUE Statement".

***cursor_for_loop_statement***

See "Cursor FOR LOOP Statement".

***execute_immediate_statement***

See "EXECUTE IMMEDIATE Statement".

***exit_statement***

See "EXIT Statement".

***fetch_statement***

See "FETCH Statement".

***for_loop_statement***

See "FOR LOOP Statement".

***forall_statement***

See "FORALL Statement".

***goto_statement***

See "GOTO Statement".

***if_statement***

See "IF Statement".

***null_statement***

See "NULL Statement".

***open_statement***

See "OPEN Statement".

***open_for_statement***

See "OPEN FOR Statement".

***pipe_row_statement***

See "PIPE ROW Statement".

**Restriction on *pipe_row_statement***

This statement can appear only in the body of a pipelined table function; otherwise, PL/SQL raises an exception.

***raise_statement***

See "RAISE Statement".

***return_statement***

See "RETURN Statement".

***select_into_statement***

See "SELECT INTO Statement".

***while_loop_statement***

See "WHILE LOOP Statement".

***procedure_call***

***procedure***

Name of the procedure that you are invoking.

***parameter* [, *parameter* ]...**

List of actual parameters for the procedure that you are invoking. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter. The mode of the formal parameter determines what the actual parameter can be:

| Formal Parameter Mode | Actual Parameter |
|---|---|
| `IN` | Constant, initialized variable, literal, or expression |
| `OUT` | Variable whose data type is not defined as `NOT NULL` |
| `IN OUT` | Variable (typically, it is a string buffer or numeric accumulator) |

If the procedure specifies a default value for a parameter, you can omit that parameter from the parameter list. If the procedure has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

> **See Also:**
>
> "Positional, Named, and Mixed Notation for Actual Parameters"

***sql_statement***

***commit_statement***

SQL `COMMIT` statement. For syntax, see *Oracle Database SQL Language Reference*.

***delete_statement***

SQL `DELETE` statement. For syntax, see *Oracle Database SQL Language Reference*. See also "DELETE Statement Extension".

***insert_statement***

SQL `INSERT` statement. For syntax, see *Oracle Database SQL Language Reference*. See also "INSERT Statement Extension".

***lock_table_statement***

SQL `LOCK TABLE` statement. For syntax, see *Oracle Database SQL Language Reference*.

***merge_statement***

SQL `MERGE` statement. For syntax, see *Oracle Database SQL Language Reference*.

***rollback_statement***

SQL `ROLLBACK` statement. For syntax, see *Oracle Database SQL Language Reference*.

***savepoint_statement***

SQL `SAVEPOINT` statement. For syntax, see *Oracle Database SQL Language Reference*.

***set_transaction_statement***

SQL `SET TRANSACTION` statement. For syntax, see *Oracle Database SQL Language Reference*.

***update_statement***

SQL `UPDATE` statement. For syntax, see *Oracle Database SQL Language Reference*. See also "UPDATE Statement Extensions".

**Examples**

- Example 2-1, "PL/SQL Block Structure"
- Example 3-23, "Block with Multiple and Duplicate Labels"

**Related Topics**

- "Comment"
- "Blocks"
- "Identifiers"
- "Pragmas"
- "PL/SQL Data Types"
- "User-Defined PL/SQL Subtypes"

# Call Specification

A **call specification** declares a Java method, C language subprogram, or JavaScript function (either exported by a Multilingual Engine (MLE) module or declared inline as part of the `CREATE FUNCTION` and `CREATE PROCEDURE` DDL statements) so that it can be invoked from PL/SQL. You can also use the SQL `CALL` statement to invoke such a method or subprogram.

The call specification tells the database which JavaScript function, Java method, or which named subprogram in which shared library, to invoke when an invocation is made. It also tells the database what type conversions to make for the arguments and return value.

A **call specification** can appear in the following SQL statements:

- ALTER TYPE Statement
- CREATE FUNCTION Statement
- CREATE PROCEDURE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

### Topics

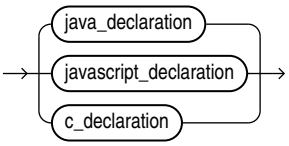
- Syntax
- Semantics
- Examples
- Related Topics

### Prerequisites

To invoke a call specification, you may need additional privileges, for example, `EXECUTE` privileges on a C library for a C call specification.

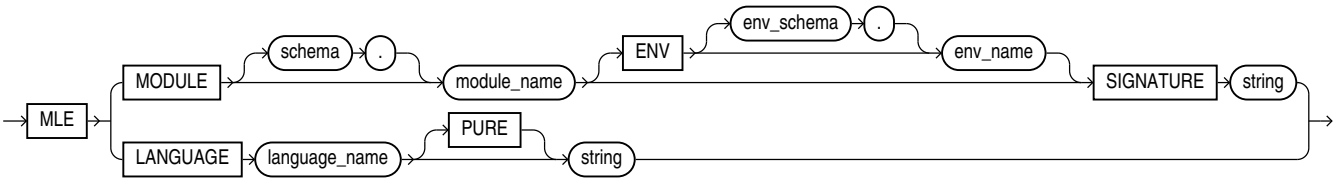
### Syntax

***call_spec* ::=**

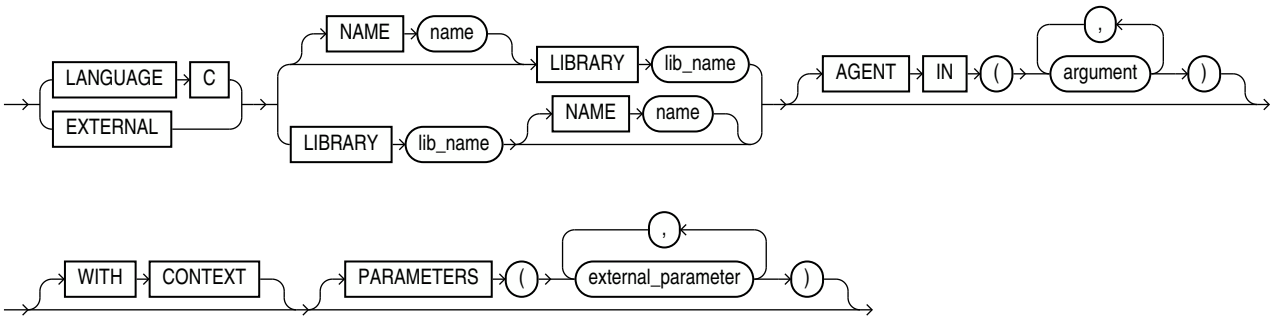



***java_declaration* ::=**

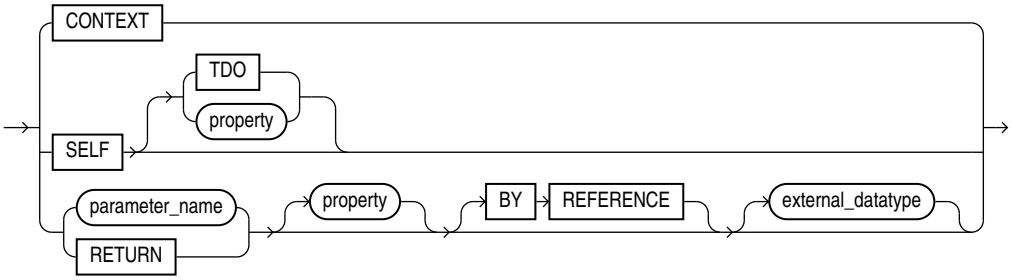

***javascript_declaration* ::=**



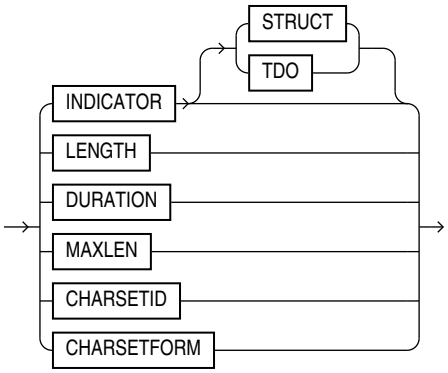

***c_declaration* ::=**



***external_parameter* ::=**




***property* ::=**




### Semantics

#### *call_spec*

Maps a C procedure, Java method name, or JavaScript function name, parameter types, and return type to their SQL counterparts.

Call specifications can appear in PL/SQL standalone subprograms, package specifications and bodies, and type specifications and bodies. They cannot appear inside PL/SQL blocks.

#### *java_declaration*

#### *string*

Identifies the Java implementation of the method.

#### *javascript_declaration*

#### *string*

Identifies the JavaScript implementation of the function.

#### *c_declaration*

#### `LIBRARY` *lib_name*

Identifies a library created by the "CREATE LIBRARY Statement".

### EXTERNAL

Deprecated way of declaring a C subprogram, supported only for backward compatibility. Use `EXTERNAL` in a C call specification if it contains defaulted arguments or constrained PL/SQL types, otherwise use the `LANGUAGE C` syntax.

### Examples

#### Example 14-6 External Function Example

The hypothetical following statement creates a PL/SQL standalone function `get_val` that registers the C subprogram `c_get_val` as an external function. (The parameters have been omitted from this example.)

```
CREATE FUNCTION get_val
   ( x_val IN NUMBER,
    y_val IN NUMBER,
    image IN LONG RAW )
   RETURN BINARY_INTEGER AS LANGUAGE C
      NAME "c_get_val"
      LIBRARY c_utils
      PARAMETERS (...);
```

### Related Topics

In this chapter:

- Function Declaration and Definition
- Procedure Declaration and Definition

In other chapters:

- CREATE LIBRARY Statement
- External Subprograms

In other books:

- *Oracle Database SQL Language Reference* for information about the `CALL` statement
- *Oracle Database Development Guide* for information about restrictions on user-defined functions that are called from SQL statements
- *Oracle Database Java Developer's Guide* to learn how to write Java call specifications
- *Oracle Database Development Guide* to learn how to write C call specifications
- *Oracle Database JavaScript Developer's Guide* to learn how to write JavaScript call specifications

# CASE Statement

The `CASE` statement chooses from a sequence of conditions and runs a corresponding statement.

The simple `CASE` statement evaluates a single expression and compares it to several potential values or expressions.

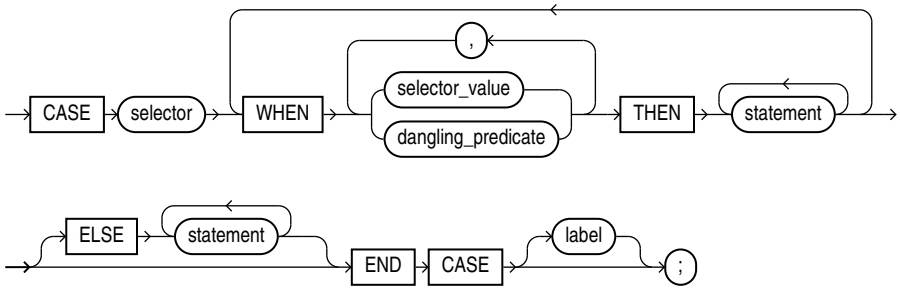
The searched `CASE` statement evaluates multiple Boolean expressions and chooses the first one whose value is `TRUE`.

### Topics

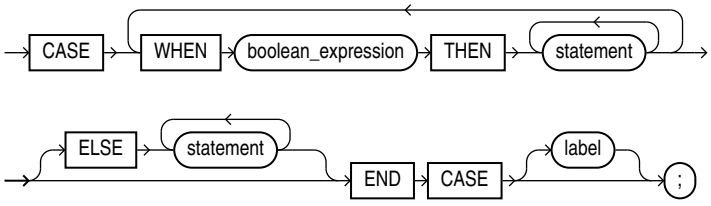
- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***simple_case_statement* ::=**




***searched_case_statement* ::=**




(*boolean_expression* ::=, *statement* ::=)

### Semantics

***simple_case_statement***

***selector***

Expression whose value is evaluated once and used to select one of several alternatives. *selector* can have any PL/SQL data type except BLOB, BFILE, or a user-defined type.

**WHEN { *selector_value* | *dangling_predicate* }**

**[ , ..., { *selector_value* | *dangling_predicate* } ] THEN *statement***

*selector_value* can be an expression of any PL/SQL type except BLOB, BFILE, or a user-defined type.

The *selector_values* and *dangling_predicates* are evaluated sequentially. If the value of a *selector_value* equals the value of *selector* or a *dangling_predicate* is true, then the *statement* associated with that *selector_value* or *dangling_predicate* runs, and the CASE

statement ends. Any subsequent *selector_values* and *dangling_predicates* are not evaluated.

> **Caution:**
>
> A *statement* can modify the database and invoke nondeterministic functions. There is no fall-through mechanism, as there is in the C `switch` statement.

> **Note:**
>
> Currently, the dangling predicates `IS JSON` and `IS OF` are not supported.

**ELSE *statement* [*statement* ]...**

The *statement*s run if and only if no *selector_value* has the same value as *selector* and no *dangling_predicate* is `true`.

Without the `ELSE` clause, if no *selector_value* has the same value as *selector* and no *dangling_predicate* is true, the system raises the predefined exception `CASE_NOT_FOUND`.

***label***

A label that identifies the statement (see "*statement* ::=" and "*label*").

***searched_case_statement***

**WHEN *boolean_expression* THEN *statement***

The *boolean_expression*s are evaluated sequentially. If the value of a *boolean_expression* is `TRUE`, the *statement* associated with that *boolean_expression* runs, and the `CASE` statement ends. Subsequent *boolean_expression*s are not evaluated.

> **Caution:**
>
> A *statement* can modify the database and invoke nondeterministic functions. There is no fall-through mechanism, as there is in the C `switch` statement.

**ELSE *statement* [*statement* ]...**

The *statement*s run if and only if no *boolean_expression* has the value `TRUE`.

Without the `ELSE` clause, if no *boolean_expression* has the value `TRUE`, the system raises the predefined exception `CASE_NOT_FOUND`.

***label***

A label that identifies the statement (see "*statement* ::=" and "*label*").

**Examples**

- Example 5-6, "Simple CASE Statement"

- Example 5-8, "Searched CASE Statement"

**Related Topics**

In this chapter:

- "IF Statement"

In other chapters:

- "CASE Expressions"
- "Conditional Selection Statements"
- "Simple CASE Statement"
- "Searched CASE Statement"

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the `NULLIF` function
> - *Oracle Database SQL Language Reference* for information about the `COALESCE` function

# CLOSE Statement

The `CLOSE` statement closes a named cursor, freeing its resources for reuse.

After closing an explicit cursor, you can reopen it with the `OPEN` statement. You must close an explicit cursor before reopening it.

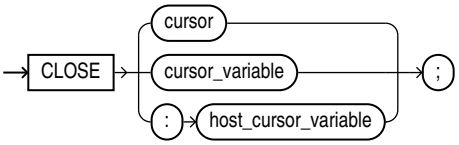
After closing a cursor variable, you can reopen it with the `OPEN FOR` statement. You need not close a cursor variable before reopening it.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***close_statement* ::=**



### Semantics

***close_statement***

***cursor***

Name of an open explicit cursor.

***cursor_variable***

Name of an open cursor variable.

**:*host_cursor_variable***

Name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *host_cursor_variable*.

### Examples

- Example 7-6, "FETCH Statements Inside LOOP Statements"

### Related Topics

In this chapter:

- "FETCH Statement"
- "OPEN Statement"
- "OPEN FOR Statement"

In other chapters:

- "Opening and Closing Explicit Cursors"
- "Opening and Closing Cursor Variables"

# Collection Method Invocation

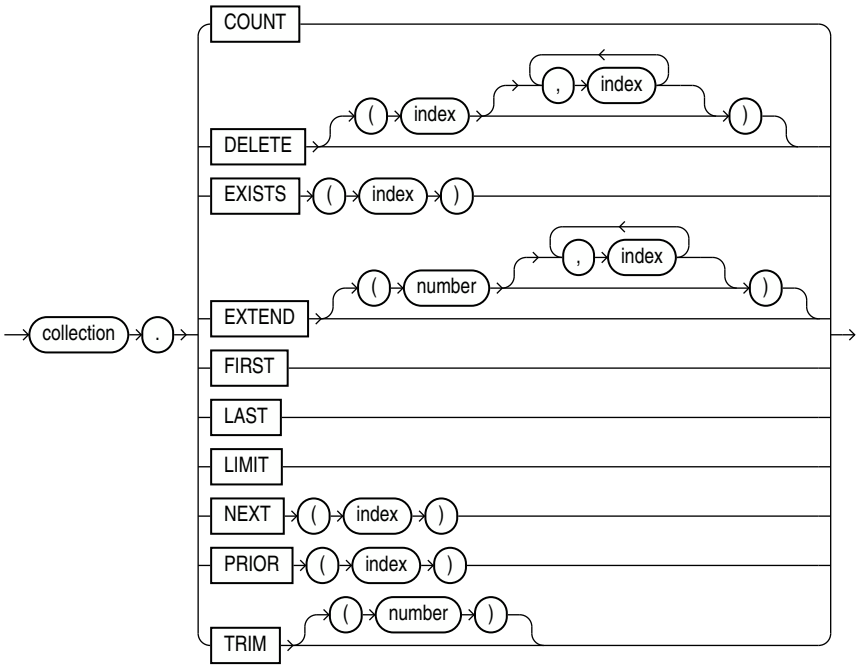
A collection method is a PL/SQL subprogram that either returns information about a collection or operates on a collection.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***collection_method_call* ::=**




### Semantics

***collection_method_call***

***collection***

Name of the collection whose method you are invoking.

**COUNT**

Function that returns the number of elements in the collection, explained in "COUNT Collection Method".

**DELETE**

Procedure that deletes elements from the collection, explained in "DELETE Collection Method".

**Restriction on DELETE**

If *collection* is a varray, you cannot specify indexes with `DELETE`.

***index***

Numeric expression whose data type either is `PLS_INTEGER` or can be implicitly converted to `PLS_INTEGER` (for information about the latter, see "s").

**EXISTS**

Function that returns `TRUE` if the *index*th element of the collection exists and `FALSE` otherwise, explained in "EXISTS Collection Method".

**EXTEND**

Procedure that adds elements to the end of the collection, explained in "EXTEND Collection Method".

**Restriction on EXTEND**

You cannot use `EXTEND` if *`collection`* is an associative array.

**FIRST**

Function that returns the first index in the collection, explained in "FIRST and LAST Collection Methods".

**LAST**

Function that returns the last index in the collection, explained in "FIRST and LAST Collection Methods".

**LIMIT**

Function that returns the maximum number of elements that the collection can have. If the collection has no maximum size, then `LIMIT` returns `NULL`. For an example, see "LIMIT Collection Method".

**NEXT**

Function that returns the index of the succeeding existing element of the collection, if one exists. Otherwise, `NEXT` returns `NULL`. For more information, see "PRIOR and NEXT Collection Methods".

**PRIOR**

Function that returns the index of the preceding existing element of the collection, if one exists. Otherwise, `NEXT` returns `NULL`. For more information, see "PRIOR and NEXT Collection Methods".

**TRIM**

Procedure that deletes elements from the end of a collection, explained in "TRIM Collection Method".

**Restriction on TRIM**

You cannot use `TRIM` if *`collection`* is an associative array.

***number***

Number of elements to delete from the end of a collection. **Default:** one.

**Examples**

- Example 6-23, "DELETE Method with Nested Table"
- Example 6-24, "DELETE Method with Associative Array Indexed by String"
- Example 6-25, "TRIM Method with Nested Table"
- Example 6-26, "EXTEND Method with Nested Table"
- Example 6-27, "EXISTS Method with Nested Table"
- Example 6-28, "FIRST and LAST Values for Associative Array Indexed by PLS_INTEGER"
- Example 6-29, "FIRST and LAST Values for Associative Array Indexed by String"

- Example 6-30, "Printing Varray with FIRST and LAST in FOR LOOP"
- Example 6-31, "Printing Nested Table with FIRST and LAST in FOR LOOP"
- Example 6-32, "COUNT and LAST Values for Varray"
- Example 6-33, "COUNT and LAST Values for Nested Table"
- Example 6-34, "LIMIT and COUNT Values for Different Collection Types"
- Example 6-35, "PRIOR and NEXT Methods"
- Example 6-36, "Printing Elements of Sparse Nested Table"

**Related Topics**

In this chapter:

- "Collection Variable Declaration"

In other chapters:

- "Collection Methods"

# Collection Variable Declaration

A **collection variable** is a composite variable whose internal components, called elements, have the same data type.

The value of a collection variable and the values of its elements can change.

You reference an entire collection by its name. You reference a collection element with the syntax *`collection(index)`*.

PL/SQL has three kinds of collection types:

- Associative array (formerly called *PL/SQL table* or *index-by table*)
- Variable-size array (varray)
- Nested table

An associative array can be indexed by either a string type or `PLS_INTEGER`. Varrays and nested tables are indexed by integers.

You can create a collection variable in either of these ways:

- Define a collection type and then declare a variable of that type.
- Use `%TYPE` to declare a collection variable of the same type as a previously declared collection variable.

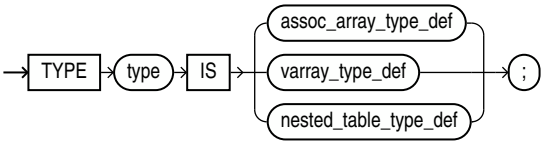
> **Note:**
>
> This topic applies to collection types that you define inside a PL/SQL block or package, which differ from standalone collection types that you create with the "CREATE TYPE Statement".
>
> In a PL/SQL block or package, you can define all three collection types. With the `CREATE TYPE` statement, you can create nested table types and `VARRAY` types, but not associative array types.

### Topics

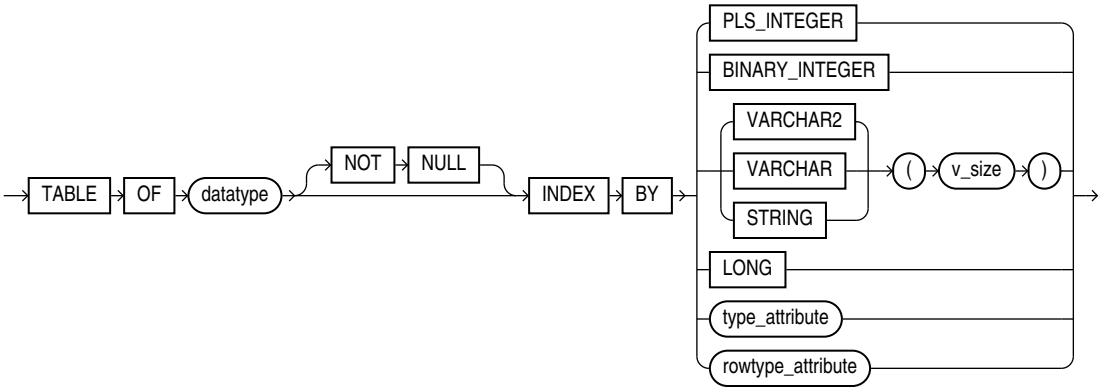
- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***collection_type_definition* ::=**

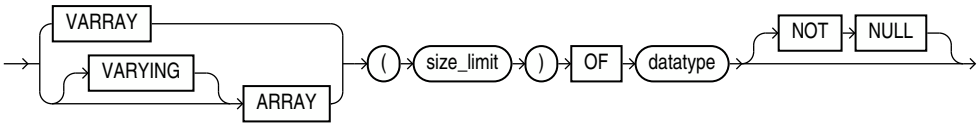



***assoc_array_type_def* ::=**



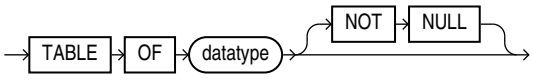

(*datatype* ::=, *rowtype_attribute* ::=, *type_attribute* ::=)

***varray_type_def* ::=**



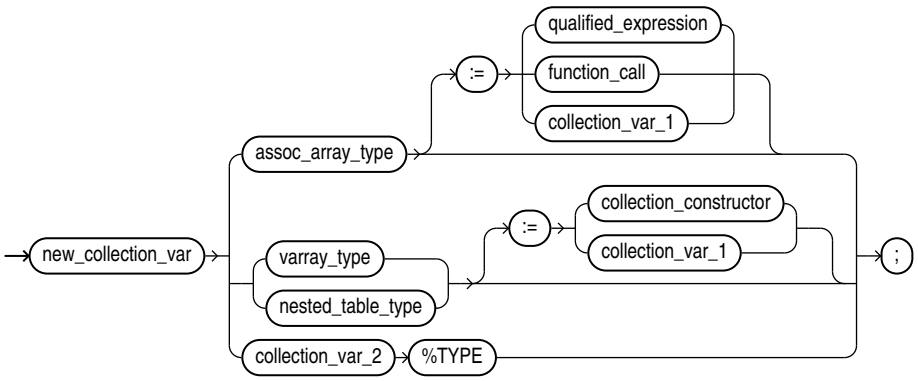

(*datatype* ::=)

***nested_table_type_def* ::=**



(*datatype* ::=)

***collection_variable_decl* ::=**




(*collection_constructor* ::=, *function_call* ::=, *qualified_expression* ::=)

**Semantics**

***collection_type_definition***

***type***

Name of the collection type that you are defining.

***assoc_array_type_def***

Type definition for an associative array.

**Restriction on *assoc_array_type_def***

Can appear only in the declarative part of a block, subprogram, package specification, or package body.

***datatype***

Data type of the elements of the associative array. *datatype* can be any PL/SQL data type except `REF CURSOR`.

**NOT NULL**

Imposes the `NOT NULL` constraint on every element of the associative array. For information about this constraint, see "NOT NULL Constraint".

**{ PLS_INTEGER | BINARY_INTEGER }**

Specifies that the data type of the indexes of the associative array is `PLS_INTEGER`.

**{ VARCHAR2 | VARCHAR | STRING } (*v_size*)**

Specifies that the data type of the indexes of the associative array is `VARCHAR2` (or its subtype `VARCHAR` or `STRING`) with length *v_size*.

You can populate an element of the associative array with a value of any type that can be converted to `VARCHAR2` with the `TO_CHAR` function (described in *Oracle Database SQL Language Reference*).

> **Caution:**
>
> Associative arrays indexed by strings can be affected by National Language Support (NLS) parameters. For more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String".

**LONG**

Specifies that the data type of the indexes of the associative array is `LONG`, which is equivalent to `VARCHAR2(32760)`.

> **Note:**
>
> Oracle supports `LONG` only for backward compatibility with existing applications. For new applications, use `VARCHAR2(32760)`.

***type_attribute, rowtype_attribute***

Specifies that the data type of the indexes of the associative array is a data type specified with either `%ROWTYPE` or `%TYPE`. This data type must represent either `PLS_INTEGER`, `BINARY_INTEGER`, or `VARCHAR2(`*v_size*`)`.

***varray_type_def***

Type definition for a variable-size array.

***size_limit***

Maximum number of elements that the varray can have. *`size_limit`* must be an integer literal in the range from 1 through 2147483647.

***datatype***

Data type of the varray element. *`datatype`* can be any PL/SQL data type except `REF CURSOR`.

**NOT NULL**

Imposes the `NOT NULL` constraint on every element of the varray. For information about this constraint, see "NOT NULL Constraint".

***nested_table_type_def***

Type definition for a nested table.

***datatype***

Data type of the elements of the nested table. *`datatype`* can be any PL/SQL data type except `REF CURSOR` or `NCLOB`.

If *`datatype`* is a scalar type, then the nested table has a single column of that type, called `COLUMN_VALUE`.

If *`datatype`* is an ADT, then the columns of the nested table match the name and attributes of the ADT.

**NOT NULL**

Imposes the `NOT NULL` constraint on every element of the nested table. For information about this constraint, see "NOT NULL Constraint".

***collection_variable_decl***

***new_collection_var***

Name of the collection variable that you are declaring.

***assoc_array_type***

Name of a previously defined associative array type; the data type of *new_collection_var*.

***varray_type***

Name of a previously defined `VARRAY` type; the data type of *new_collection_var*.

***nested_table_type***

Name of a previously defined nested table type; the data type of *new_collection_var*.

***collection_constructor***

Collection constructor for the data type of *new_collection_var*, which provides the initial value of *new_collection_var*.

***collection_var_1***

Name of a previously declared collection variable of the same data type as *new_collection_var*, which provides the initial value of *new_collection_var*.

> **Note:**
>
> *collection_var_1* and *new_collection_var* must have the same data type, not only elements of the same type.

***collection_var_2***

Name of a previously declared collection variable.

**%TYPE**

See "%TYPE Attribute".

**Examples**

- Example 6-1, "Associative Array Indexed by String"
- Example 6-2, "Function Returns Associative Array Indexed by PLS_INTEGER"
- Example 6-4, "Varray (Variable-Size Array)"
- Example 6-5, "Nested Table of Local Type"
- Example 6-17, "Two-Dimensional Varray (Varray of Varrays)"
- Example 6-18, "Nested Tables of Nested Tables and Varrays of Integers"

### Related Topics

- "Qualified Expressions Overview"
- "Collection Topics"
- "BULK COLLECT Clause"
- "CREATE TYPE Statement"
- "Collection Method Invocation"
- "FORALL Statement"
- "Record Variable Declaration"
- "%ROWTYPE Attribute"
- "%TYPE Attribute"

# Comment

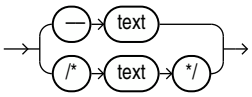
A comment is source program text that the PL/SQL compiler ignores. Its primary purpose is to document code, but you can also use it to disable obsolete or unfinished pieces of code (that is, you can turn the code into comments). PL/SQL has both single-line and multiline comments.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***comment* ::=**




### Semantics

***comment***

**--**

Turns the rest of the line into a single-line comment. Any text that wraps to the next line is not part of the comment.

> **Caution:**
>
> Do not put a single-line comment in a PL/SQL block to be processed dynamically by an Oracle Precompiler program. The Oracle Precompiler program ignores end-of-line characters, which means that a single-line comment ends when the block ends.

***/****

Begins a comment, which can span multiple lines.

****/***

Ends a comment.

***text***

Any text.

**Restriction on *text***

In a multiline comment, *text* cannot include the multiline comment delimiter `/*` or `*/`. Therefore, one multiline comment cannot contain another multiline comment. However, a multiline comment can contain a single-line comment.

**Examples**

- Example 3-6, "Single-Line Comments"
- Example 3-7, "Multiline Comments"

**Related Topics**

- "Comments"

# COMPILE Clause

The compile clause explicitly recompiles a stored unit that has become invalid, thus eliminating the need for implicit runtime recompilation and preventing associated runtime compilation errors and performance overhead.

The `COMPILE` clause can appear in the following SQL statements:

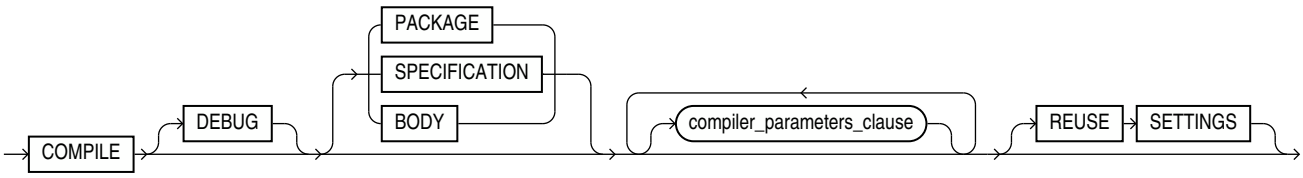
- ALTER FUNCTION Statement
- ALTER PACKAGE Statement
- ALTER PROCEDURE Statement
- ALTER LIBRARY Statement
- ALTER TYPE Statement
- ALTER TRIGGER Statement

**Topics**

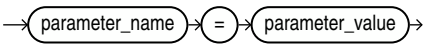
- Syntax
- Semantics
- Related Topics

### Syntax

***compile_clause* ::=**




***compiler_parameters_clause* ::=**




### Semantics

***compile_clause***

**COMPILE**

Recompiles the PL/SQL unit, whether it is valid or invalid. The PL/SQL unit can be a library, package, package specification, package body, trigger, procedure, function, type, type specification, or type body.

First, if any of the objects upon which the unit depends are invalid, the database recompiles them.

The database also invalidates any local objects that depend upon the unit.

If the database recompiles the unit successfully, then the unit becomes valid. Otherwise, the database returns an error and the unit remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

**DEBUG**

Has the same effect as `PLSQL_OPTIMIZE_LEVEL=1`—instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Oracle recommends using `PLSQL_OPTIMIZE_LEVEL=1` instead of `DEBUG`.

**PACKAGE**

**(Default)** Recompiles both the package specification and (if it exists) the package body, whether they are valid or invalid. The recompilation of the package specification and body lead to the invalidation and recompilation of dependent objects as described for `SPECIFICATION` and `BODY`.

**Restriction on `PACKAGE`**

`PACKAGE` may only appear if compiling a package.

### SPECIFICATION

Recompiles only the package or type specification, whether it is valid or invalid. You might want to recompile a package or type specification to check for compilation errors after modifying the specification.

When you recompile a specification, the database invalidates any local objects that depend on the specification, such as procedures that invoke procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

#### Restriction on `SPECIFICATION`

`SPECIFICATION` may only appear if compiling a package or type specification.

### BODY

Recompiles only the package or type body, whether it is valid or invalid. You might want to recompile a package or type body after modifying it. Recompiling a body does not invalidate objects that depend upon its specification.

When you recompile a package or type body, the database first recompiles the objects on which the body depends, if any of those objects are invalid. If the database recompiles the body successfully, then the body becomes valid.

#### Restriction on `BODY`

`BODY` may only appear if compiling a package or type body.

### REUSE SETTINGS

Prevents Oracle Database from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in this statement.

See also DEFAULT COLLATION Clause compilation semantics.

#### *compiler_parameters_clause*

Specifies a value for a PL/SQL compilation parameter in Table 2-2. The compile-time value of each of these parameters is stored with the metadata of the PL/SQL unit being compiled.

You can specify each parameter only once in each statement. Each setting is valid only for the PL/SQL unit being compiled and does not affect other compilations in this session or system. To affect the entire session or system, you must set a value for the parameter using the `ALTER SESSION` or `ALTER SYSTEM` statement.

If you omit any parameter from this clause and you specify `REUSE SETTINGS`, then if a value was specified for the parameter in an earlier compilation of this PL/SQL unit, the database uses that earlier value. If you omit any parameter and either you do not specify `REUSE SETTINGS` or no value was specified for the parameter in an earlier compilation, then the database obtains the value for that parameter from the session environment.

#### Related Topics

In other books:

- *Oracle Database Development Guide* for information about debugging procedures

- *Oracle Database Development Guide* for information about debugging a trigger using the same facilities available for stored subprograms

# Constant Declaration

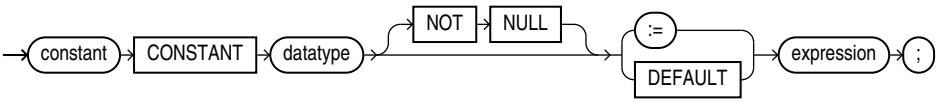
A constant holds a value that does not change. A constant declaration specifies the name, data type, and value of the constant and allocates storage for it. The declaration can also impose the `NOT NULL` constraint.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***constant_declaration* ::=**




(*datatype* ::=, *expression* ::=)

**Semantics**

***constant_declaration***

***constant***

Name of the constant that you are declaring.

***datatype***

Data type for which a variable can be declared with an initial value.

**NOT NULL**

Imposes the `NOT NULL` constraint on the constant.

For information about this constraint, see "NOT NULL Constraint".

***expression***

Initial value for the constant. *expression* must have a data type that is compatible with *datatype*. When *constant_declaration* is elaborated, the value of *expression* is assigned to *constant*.

**Examples**

- Example 3-12, "Constant Declarations"
- Example 3-13, "Variable and Constant Declarations with Initial Values"

### Related Topics

In this chapter:

- "Collection Variable Declaration"
- "Record Variable Declaration"
- "%ROWTYPE Attribute"
- "Scalar Variable Declaration"
- "%TYPE Attribute"

In other chapters:

- "Declaring Constants"
- "Declaring Associative Array Constants"
- "Declaring Record Constants"

# CONTINUE Statement

The `CONTINUE` statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

If a `CONTINUE` statement exits a cursor `FOR` loop prematurely (for example, to exit an inner loop and transfer control to the next iteration of an outer loop), the cursor closes (in this context, `CONTINUE` works like `GOTO`).

The `CONTINUE WHEN` statement exits the current iteration of a loop when the condition in its `WHEN` clause is true, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

Each time control reaches the `CONTINUE WHEN` statement, the condition in its `WHEN` clause is evaluated. If the condition is not true, the `CONTINUE WHEN` statement does nothing.

### Restrictions on CONTINUE Statement

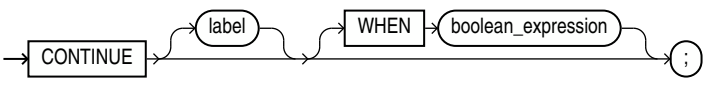
- A `CONTINUE` statement must be inside a `LOOP` statement.
- A `CONTINUE` statement cannot cross a subprogram or method boundary.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***continue_statement* ::=**



(*boolean_expression* ::=)

### Semantics

***continue_statement***

***label***

Name that identifies either the current loop or an enclosing loop.

Without *label*, the CONTINUE statement transfers control to the next iteration of the current loop. With *label*, the CONTINUE statement transfers control to the next iteration of the loop that *label* identifies.

**WHEN *boolean_expression***

Without this clause, the CONTINUE statement exits the current iteration of the loop unconditionally. With this clause, the CONTINUE statement exits the current iteration of the loop if and only if the value of *boolean_expression* is TRUE.

### Examples

#### Example 14-7 CONTINUE Statement in Basic LOOP Statement

In this example, the CONTINUE statement inside the basic LOOP statement transfers control unconditionally to the next iteration of the current loop.

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

#### Example 14-8 CONTINUE WHEN Statement in Basic LOOP Statement

In this example, the CONTINUE WHEN statement inside the basic LOOP statement transfers control to the next iteration of the current loop when x is less than 3.

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    CONTINUE WHEN x < 3;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

### Related Topics

- "LOOP Statements" for more conceptual information
- "Basic LOOP Statement" for more information about labelling loops
- "Cursor FOR LOOP Statement"
- "EXIT Statement"
- "Expression"
- "FOR LOOP Statement"
- "WHILE LOOP Statement"

# COVERAGE Pragma

The `COVERAGE` pragma marks PL/SQL code which is infeasible to test for coverage. These marks improve coverage metric accuracy.

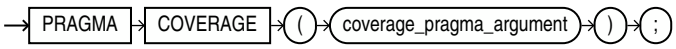
The `COVERAGE` pragma marks PL/SQL source code to indicate that the code may not be feasibly tested for coverage. The pragma marks a specific code section. Marking infeasible code improves the quality of coverage metrics used to assess how much testing has been achieved.

### Topics

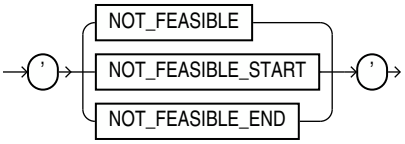
- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***coverage_pragma ::=***




***coverage_pragma_argument* ::=**




### Semantics

***coverage_pragma***

The `COVERAGE` pragma may appear before any declaration or statement.

***coverage_pragma_argument***

The `COVERAGE` pragma argument must have one of these values:

- 'NOT_FEASIBLE'
- 'NOT_FEASIBLE_START'
- 'NOT_FEASIBLE_END'

When the `COVERAGE` pragma appear with the argument 'NOT_FEASIBLE', it marks the entire basic block that includes the beginning of the first declaration or statement that follows the pragma.

A `COVERAGE` pragma with an argument of 'NOT_FEASIBLE_START' may appear before any declaration or any statement. It must be followed by the `COVERAGE` pragma with an argument of 'NOT_FEASIBLE_END'. The second pragma may appear before any declaration or any statement. It must appear in the same PL/SQL block as the first pragma and not in any nested subprogram definition.

An associated pair of `COVERAGE` pragmas marks basic blocks infeasible from the beginning of the basic block that includes the beginning of the first statement or declaration that follows the first pragma to the end of the basic block that includes the first statement or declaration that follows the second pragma.

A `COVERAGE` pragma whose range includes the definition or declaration of an inner subprogram does not mark the blocks of that subprogram as infeasible.

### Examples

#### Example 14-9 Marking a Single Basic Block as Infeasible to Test for Coverage

This example shows the placement of the pragma `COVERAGE` preceding the assignments to z and zl basic blocks. These two basic blocks will be ignored for coverage calculation. The first `COVERAGE` pragma (marked 1) marks the first assignment to z infeasible; the second (marked 2)

marks the third assignment to z. In each case, the affected basic block runs from the identifier z to the following `END IF`.

```
IF (x>0) THEN
  y :=2;
ELSE
  PRAGMA COVERAGE ('NOT_FEASIBLE');  -- 1
  z:=3;
END IF;
IF (y>0) THEN
  z :=2;
ELSE
  PRAGMA COVERAGE ('NOT_FEASIBLE'); -- 2
  z :=3;
END IF;
```

### Example 14-10 Marking a Line Range as Infeasible to Test for Coverage

This examples shows marking the entire line range as not feasible. A line range may contain more than one basic block. A line range is marked as not feasible for coverage using a pragma `COVERAGE` with a 'NOT_FEASIBLE_START' argument at the beginning of the range, and a pragma `COVERAGE` with a 'NOT_FEASIBLE_END' at the end of the range. The range paired `COVERAGE` pragmas mark all the blocks as infeasible.

```
PRAGMA COVERAGE ('NOT_FEASIBLE_START');
IF (x>0) THEN
  y :=2;
ELSE
  z:=3;
END IF;
IF (y>0) THEN
  z :=2;
ELSE
  z :=3;
END IF;
PRAGMA COVERAGE ('NOT_FEASIBLE_END');
```

### Example 14-11 Marking Entire Units or Individual Subprograms as Infeasible to Test for Coverage

This example shows marking the entire procedure foo as not feasible for coverage. A subprogram is marked as completely infeasible by marking all of its body infeasible.

```
CREATE PROCEDURE foo IS
PRAGMA COVERAGE ('NOT_FEASIBLE_START');
......

BEGIN
....
PRAGMA COVERAGE ('NOT_FEASIBLE_END');
END;
/
```

### Example 14-12 Marking Internal Subprogram as Infeasible to Test for Coverage

This example shows that the outer `COVERAGE` pragma pair has no effect on coverage inside procedure inner. The `COVERAGE` pragma (marked 1) inside the body of inner does mark the second assignment to x as infeasible. Notice that the entire body of procedure outer is marked infeasible even though the pragma with argument 'NOT_FEASIBLE_END' is not the last line. The pragma does mark the basic block that includes the statement that follows the pragma and that block does extend to the end of the procedure.

```
CREATE OR REPLACE PROCEDURE outer IS
   PRAGMA COVERAGE ('NOT_FEASIBLE_START');
   x NUMBER := 7;
   PROCEDURE inner IS
   BEGIN
     IF x < 6 THEN
        x := 19;
     ELSE
        PRAGMA COVERAGE ('NOT_FEASIBLE'); -- 1
        x := 203;
     END IF;
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE ('X= ');
   PRAGMA COVERAGE ('NOT_FEASIBLE_END');
   DBMS_OUTPUT.PUT_LINE (x);
END;
/
```

#### Related Topics

In this book:

- Pragmas
- PL/SQL Units and Compilation Parameters for more information about the PLSQL_OPTIMIZE_LEVEL compilation parameter

In other books:

- *Oracle Database Development Guide* for more information about using PL/SQL basic block coverage to maintain quality
- *Oracle Database PL/SQL Packages and Types Reference* for more information about using the DBMS_PLSQL_CODE_COVERAGE package

# Cursor FOR LOOP Statement

The cursor `FOR LOOP` statement implicitly declares its loop index as a record variable of the row type that a specified cursor returns, and then opens a cursor.

With each iteration, the cursor `FOR LOOP` statement fetches a row from the result set into the record. When there are no more rows to fetch, the cursor `FOR LOOP` statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or raises an exception.

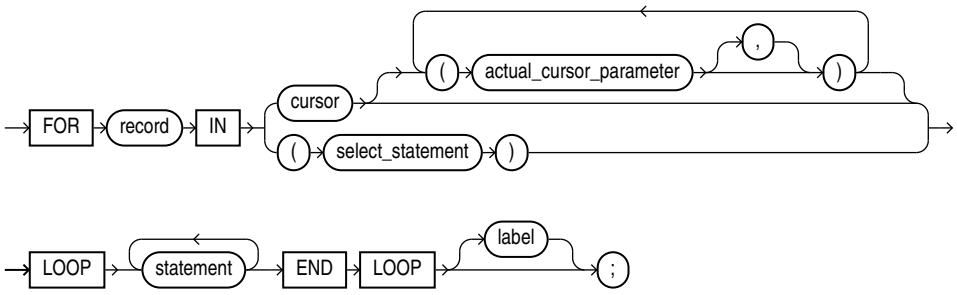
#### Topics

- Syntax

- Semantics
- Examples
- Related Topics

### Syntax

***cursor_for_loop_statement* ::=**




(*statement* ::=)

### Semantics

***cursor_for_loop_statement***

***record***

Name for the loop index that the cursor `FOR LOOP` statement implicitly declares as a `%ROWTYPE` record variable of the type that *`cursor`* or *`select_statement`* returns.

*`record`* is local to the cursor `FOR LOOP` statement. Statements inside the loop can reference *`record`* and its fields. They can reference virtual columns only by aliases. Statements outside the loop cannot reference *`record`*. After the cursor `FOR LOOP` statement runs, *`record`* is undefined.

***cursor***

Name of an explicit cursor (not a cursor variable) that is not open when the cursor `FOR LOOP` is entered.

***actual_cursor_parameter***

Actual parameter that corresponds to a formal parameter of *`cursor`*.

***select_statement***

SQL `SELECT` statement (not PL/SQL `SELECT INTO` statement). For *`select_statement`*, PL/SQL declares, opens, fetches from, and closes an implicit cursor. However, because *`select_statement`* is not an independent statement, the implicit cursor is internal—you cannot reference it with the name `SQL`.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `SELECT` statement syntax

***label***

Label that identifies *cursor_for_loop_statement* (see "*statement* ::=" and "*label*"). `CONTINUE`, `EXIT`, and `GOTO` statements can reference this label.

Labels improve readability, especially when `LOOP` statements are nested, but only if you ensure that the label in the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

**Examples**

- Example 7-18, "Implicit Cursor FOR LOOP Statement"
- Example 7-19, "Explicit Cursor FOR LOOP Statement"
- Example 7-20, "Passing Parameters to Explicit Cursor FOR LOOP Statement"
- Example 7-21, "Cursor FOR Loop References Virtual Columns"

**Related Topics**

In this chapter:

- "Basic LOOP Statement"
- "CONTINUE Statement"
- "EXIT Statement"
- "Explicit Cursor Declaration and Definition"
- "FETCH Statement"
- "FOR LOOP Statement"
- "FORALL Statement"
- "OPEN Statement"
- "WHILE LOOP Statement"

In other chapters:

- "Processing Query Result Sets With Cursor FOR LOOP Statements"

# Cursor Variable Declaration

A cursor variable is like an explicit cursor that is not limited to one query.

To create a cursor variable, either declare a variable of the predefined type `SYS_REFCURSOR` or define a `REF CURSOR` type and then declare a variable of that type.

**Restrictions on Cursor Variables**

- You cannot declare a cursor variable in a package specification.

  That is, a package cannot have a public cursor variable (a cursor variable that can be referenced from outside the package).
- You cannot store the value of a cursor variable in a collection or database column.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.

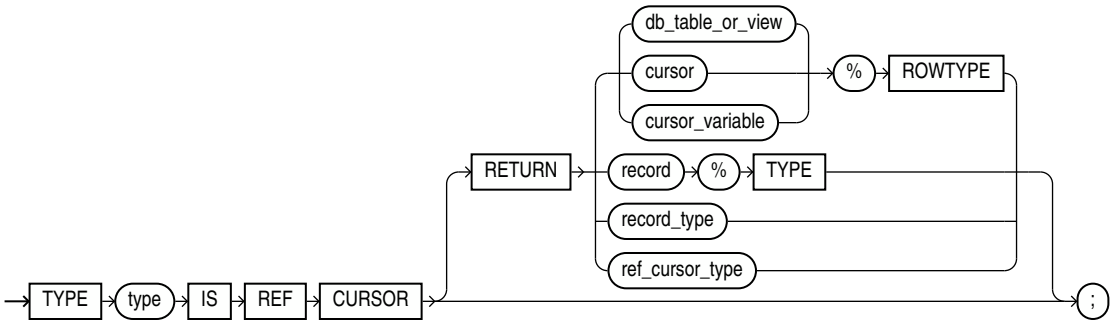
- Using a cursor variable in a server-to-server remote procedure call (RPC) causes an error. However, you can use a cursor variable in a server-to-server RPC if the remote database is a non-Oracle database accessed through a Procedural Gateway.

**Topics**

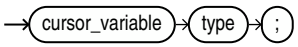

- Syntax
- Semantics
- Examples
- Related Topics


**Syntax**

***ref_cursor_type_definition* ::=**




***cursor_variable_declaration* ::=**




**Semantics**

***ref_cursor_type_definition***

***type***

Name of the `REF CURSOR` type that you are defining.

**RETURN**

Specifies the data type of the value that the cursor variable returns.

Specify `RETURN` to define a strong `REF CURSOR` type. Omit `RETURN` to define a weak `REF CURSOR` type. For information about strong and weak `REF CURSOR` types, see "Creating Cursor Variables".

***db_table_or_view***

Name of a database table or view, which must be accessible when the declaration is elaborated.

***cursor***

Name of a previously declared explicit cursor.

***cursor_variable***

Name of a previously declared cursor variable.

***record***

Name of a user-defined record.

***record_type***

Name of a user-defined type that was defined with the data type specifier `RECORD`.

***ref_cursor_type***

Name of a user-defined type that was defined with the data type specifier `REF CURSOR`.

***cursor_variable_declaration***

***cursor_variable***

Name of the cursor variable that you are declaring.

***type***

Type of the cursor variable that you are declaring—either `SYS_REFCURSOR` or the name of the `REF CURSOR` type that you defined previously.

`SYS_REFCURSOR` is a weak type. For information about strong and weak `REF CURSOR` types, see "Creating Cursor Variables".

**Examples**

- Example 7-24, "Cursor Variable Declarations"
- Example 7-25, "Cursor Variable with User-Defined Return Type"
- Example 7-28, "Variable in Cursor Variable Query—No Result Set Change"
- Example 7-29, "Variable in Cursor Variable Query—Result Set Change"
- Example 7-30, "Querying a Collection with Static SQL"
- Example 7-31, "Procedure to Open Cursor Variable for One Query"
- Example 7-32, "Opening Cursor Variable for Chosen Query (Same Return Type)"
- Example 7-33, "Opening Cursor Variable for Chosen Query (Different Return Types)"
- Example 7-34, "Cursor Variable as Host Variable in Pro*C Client Program"

**Related Topics**

In this chapter:

- "CLOSE Statement"
- "Named Cursor Attribute"
- "Explicit Cursor Declaration and Definition"
- "FETCH Statement"
- "OPEN FOR Statement"
- "%ROWTYPE Attribute"
- "%TYPE Attribute"

In other chapters:

- "Cursor Variables"
- "Passing CURSOR Expressions to Pipelined Table Functions"

# Datatype Attribute

The data type attribute of an ADT element.

A `datatype` allows you to declare the data type of record variables fields, constants, functions return value, collection variables and collection types elements.

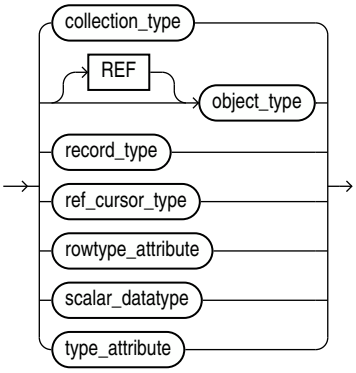
- Record Variable Declaration
- Constant Declaration
- Function Declaration and Definition
- Collection Variable Declaration
- CREATE FUNCTION Statement
- CREATE TYPE Statement

**Topics**

- Syntax
- Semantics
- Related Topics

**Syntax**

***datatype* ::=**




(*rowtype_attribute* ::=, *type_attribute* ::=)

**Semantics**

***datatype***

***collection_type***

Name of a user-defined varray or nested table type (not the name of an associative array type).

***object_type***

Instance of a user-defined type.

***record_type***

Name of a user-defined type that was defined with the data type specifier `RECORD`.

***ref_cursor_type***

Name of a user-defined type that was defined with the data type specifier `REF CURSOR`.

***scalar_datatype***

Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics.

**Related Topics**

In other chapters:

- PL/SQL Data Types

# DEFAULT COLLATION Clause

Collation (also called sort ordering) determines if a character string equals, precedes, or follows another string when the two strings are compared and sorted. Oracle Database collations order strings following rules for sorted text used in different languages.

The `DEFAULT COLLATION` clause can appear in the following SQL statements:

- CREATE FUNCTION Statement
- CREATE PROCEDURE Statement
- CREATE PACKAGE Statement
- CREATE TRIGGER Statement
- CREATE TYPE Statement

**Topics**

- Prerequisites
- Syntax
- Semantics
- Compilation Semantics
- Related Topics

**Prerequisites**

The `COMPATIBLE` initialization parameter must be set to at least 12.2.0, and `MAX_STRING_SIZE` must be set to `EXTENDED` for collation declarations to be allowed in these SQL statements.

**Syntax**

***default_collation_clause*** **::=**

***collation_option* ::=**


### Semantics

#### *default_collation_clause*

The *default_collation_clause* can appear in a package specification, a standalone type specification, and in standalone subprograms.

#### *collation_option*

The default collation of a procedure, function, package, type, or trigger must be `USING_NLS_COMP`. The *default_collation_clause* explicitly declares the default collation of a PL/SQL unit to be `USING_NLS_COMP`. Without this clause, the unit inherits its default collation from the effective schema default collation. If the effective schema default collation is not `USING_NLS_COMP`, the unit is invalid.

The effective schema default collation is determined as follows:

- If the session parameter `DEFAULT_COLLATION` is set, the effective schema default collation is the value of this parameter. The value of the parameter can be checked by querying `SYS_CONTEXT('USERENV', 'SESSION_DEFAULT_COLLATION')`. The function returns `NULL` if `DEFAULT_COLLATION` is not set. The value of the parameter `DEFAULT_COLLATION` can be set with the statement: `ALTER SESSION SET DEFAULT_COLLATION` = *collation_option*;
- If the session parameter `DEFAULT_COLLATION` is not set, the effective schema default collation is the declared default collation of the schema in which you create the PL/SQL unit. The default collation of a schema can be found in the static data dictionary *_USERS views. It can be set with the DDL statements `CREATE USER` and `ALTER USER`.

The session parameter `DEFAULT_COLLATION` can be unset with the statement: `ALTER SESSION SET DEFAULT_COLLATION = NONE;`

Package body and type body use the default collation of the corresponding specification. All character data containers and attributes in procedures, functions and methods, including parameters and return values, behave as if their data-bound collation were the pseudo-collation `USING_NLS_COMP`.

#### Restrictions on `DEFAULT COLLATION`

It cannot be specified for nested or packaged subprograms or for type methods.

#### Compilation Semantics

If the resulting default object collation is different from `USING_NLS_COMP`, the database object is created as invalid with a compilation error.

If the `ALTER COMPILE` statement is issued for a PL/SQL unit with the `REUSE SETTINGS` clause, the stored default collation of the database object being compiled is not changed.

If an `ALTER COMPILE` statement is issued without the `REUSE SETTINGS` clause, the stored default collation of the database object being compiled is discarded and the effective schema default collation for the object owner at the time of execution of the statement is stored as the default

collation of the object, unless the PL/SQL unit contains the `DEFAULT COLLATION` clause. If the resulting default collation is not `USING_NLS_COMP`, a compilation error is raised.

An `ALTER COMPILE` statement for a package or type body references the stored collation of the corresponding specification.

**Related Topics**

In other chapters:

- ALTER FUNCTION Statement
- ALTER PACKAGE Statement
- ALTER PROCEDURE Statement
- ALTER TRIGGER Statement
- ALTER TYPE Statement

In other books :

- *Oracle Database Globalization Support Guide* for more information about specifying data-bound collation for PL/SQL units
- *Oracle Database Globalization Support Guide* for more information about effective schema default collation

# DELETE Statement Extension

The PL/SQL extension to the *`where_clause`* of the SQL `DELETE` statement lets you specify a `CURRENT OF` clause, which restricts the `DELETE` statement to the current row of the specified cursor.

For information about the `CURRENT OF` clause, see "UPDATE Statement Extensions".

> **See Also:**
>
> *Oracle Database SQL Language Reference* for the syntax of the SQL `DELETE` statement

# DEPRECATE Pragma

The `DEPRECATE` pragma marks a PL/SQL element as deprecated. The compiler issues warnings for uses of pragma `DEPRECATE` or of deprecated elements.

The associated warnings tell users of a deprecated element that other code may need to be changed to account for the deprecation.

**Topics**

- Syntax
- Semantics
- DEPRECATE Pragma Compilation Warnings
- Examples

- Related Topics

### Syntax

***deprecate_pragma ::=***


### Semantics

***deprecate_pragma***

The `DEPRECATE` pragma may only appear in the declaration sections of a package specification, an object specification, a top level procedure, or a top level function.

PL/SQL elements of these kinds may be deprecated:

- Subprograms
- Packages
- Variables
- Constants
- Types
- Subtypes
- Exceptions
- Cursors

The `DEPRECATE` pragma may only appear in the declaration section of a PL/SQL unit. It must appear immediately after the declaration of an item to be deprecated.

The `DEPRECATE` pragma applies to the PL/SQL element named in the declaration which precedes the pragma.

When the `DEPRECATE` pragma applies to a package specification, object specification, or subprogram, the pragma must appear immediately after the keyword `IS` or `AS` that terminates the declaration portion of the definition.

When the `DEPRECATE` pragma applies to a package or object specification, references to all the elements (of the kinds that can be deprecated) that are declared in the specification are also deprecated.

If the `DEPRECATE` pragma applies to a subprogram declaration, only that subprogram is affected; other overloads with the same name are not deprecated.

If the optional custom message appears in a use of the `DEPRECATE` pragma, the custom message will be added to the warning issued for any reference to the deprecated element.

The identifier in a `DEPRECATE` pragma must name the element in the declaration to which it applies.

Deprecation is inherited during type derivation. A child object type whose parent is deprecated is not deprecated. Only the attributes and methods that are inherited are deprecated.

When the base type is not deprecated but individual methods or attributes are deprecated, and when a type is derived from this type and the deprecated type or method is inherited, then references to these through the derived type will cause the compiler to issue a warning.
A reference to a deprecated element appearing anywhere except in the unit with the deprecation pragma or its body, will cause the PL/SQL compiler to issue a warning for the referenced elements. A reference to a deprecated element in an anonymous block will not cause the compiler to issue a warning; only references in named entities will draw a warning.

When a deprecated entity is referenced in the definition of another deprecated entity then no warning will be issued.

When an older client code refers to a deprecated entity, it is invalidated and recompiled. No warning is issued.

There is no effect when SQL code directly references a deprecated element.

A reference to a deprecated element in a PL/SQL static SQL statement may cause the PL/SQL compiler to issue a warning. However, such references may not be detectable.

***pls_identifier***

Identifier of the PL/SQL element being deprecated.

***character_literal***

An optional compile-time warning message.

### DEPRECATE Pragma Compilation Warnings

The PL/SQL compiler issues warnings when the `DEPRECATE` pragma is used and when deprecated items are referenced.

- 6019 — The entity was deprecated and could be removed in a future release. Do not use the deprecated entity.
- 6020 — The referenced entity was deprecated and could be removed in a future release. Do not use the deprecated entity. Follow the specific instructions in the warning if any are given.
- 6021 — Misplaced pragma. The pragma `DEPRECATE` should follow immediately after the declaration of the entity that is being deprecated. Place the pragma immediately after the declaration of the entity that is being deprecated.
- 6022 — This entity cannot be deprecated. Deprecation only applies to entities that may be declared in a package or type specification as well as to top-level procedure and function definitions. Remove the pragma.

The `DEPRECATE` pragma warnings may be managed with the `PLSQL_WARNINGS` parameter or with the DBMS_WARNING package.

### Examples

#### Example 14-13 Enabling the Deprecation Warnings

This example shows how to set the PLSQL_WARNINGS parameter to enable these warnings in a session.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Restricting Access to Top-Level Procedures in the Same Schema

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:(6019,6020,6021,6022)';
```

### Example 14-14 Deprecation of a PL/SQL Package

This example shows the deprecation of a PL/SQL package as a whole. Warnings will be issued for any reference to package pack1, and to the procedures foo and bar when used outside of the package and its body.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of a PL/SQL Package

```
PACKAGE pack1 AS
PRAGMA DEPRECATE(pack1);
 PROCEDURE foo;
 PROCEDURE bar;
END pack1;
```

### Example 14-15 Deprecation of a PL/SQL Package with a Custom Warning

This example shows the deprecation of a PL/SQL package. The compiler issues a custom warning message when a reference in another unit for the deprecated procedure foo is compiled.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of a PL/SQL Package with a Custom Warning

```
PACKAGE pack5 AUTHID DEFINER AS
PRAGMA DEPRECATE(pack5 , 'Package pack5 has been deprecated, use new_pack5
instead.');
 PROCEDURE foo;
 PROCEDURE bar;
END pack5;
```

A reference to procedure pack5.foo in another unit would draw a warning like this.

```
SP2-0810: Package Body created with compilation warnings

Errors for PACKAGE BODY PACK6:
```

```
4/10     PLW-06020: reference to a deprecated entity: PACK5 declared in unit
PACK5[1,9].
         Package pack5 has been deprecated, use new_pack5 instead
```

### Example 14-16 Deprecation of a PL/SQL Procedure

This example shows the deprecation of a single PL/SQL procedure foo in package pack7.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of a PL/SQL Procedure

```
PACKAGE pack7 AUTHID DEFINER AS
  PROCEDURE foo;
  PRAGMA DEPRECATE (foo, 'pack7.foo is deprecated, use pack7.bar instead.');
  PROCEDURE bar;
END pack7;
```

### Example 14-17 Deprecation of an Overloaded Procedure

This example shows the `DEPRECATE` pragma applies only to a specific overload of a procedure name. Only the second declaration of proc1 is deprecated.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of an Overloaded Procedure

```
PACKAGE pack2 AS
  PROCEDURE proc1(n1 NUMBER, n2 NUMBER, n3 NUMBER);
  -- Only the overloaded procedure with 2 arguments is deprecated
  PROCEDURE proc1(n1 NUMBER, n2 NUMBER);
      PRAGMA DEPRECATE(proc1);
 END pack2;
```

### Example 14-18 Deprecation of a Constant and of an Exception

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of a Constant and of an Exception

This example shows the deprecation of a constant and of an exception.

```
PACKAGE trans_data AUTHID DEFINER AS
   TYPE Transrec IS RECORD (
```

```
    accounttype VARCHAR2(30) ,
    ownername VARCHAR2(30) ,
    balance REAL
  );
  min_balance constant real := 10.0;
  PRAGMA DEPRECATE(min_balance , 'Minimum balance requirement has been
removed.');
  insufficient_funds EXCEPTION;
  PRAGMA DEPRECATE (insufficient_funds , 'Exception no longer raised.');
END trans_data;
```

### Example 14-19 Using Conditional Compilation to Deprecate Entities in Some Database Releases

This example shows the deprecation of procedure proc1 if the database release version is greater than 11.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Using Conditional Compilation to Deprecate Entities in Some Database Releases

```
CREATE PACKAGE pack11 AUTHID DEFINER AS
 $IF DBMS_DB_VERSION.VER_LE_11
 $THEN
    PROCEDURE proc1;
 $ELSE
    PROCEDURE proc1;
    PRAGMA DEPRECATE(proc1);
 $END
 PROCEDURE proc2;
 PROCEDURE proc3;
END pack11;
```

### Example 14-20 Deprecation of an Object Type

This example shows the deprecation of an entire object type.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of an Object Type

```
TYPE type01 AS OBJECT(
   PRAGMA DEPRECATE (type01),
   y NUMBER,
   MEMBER PROCEDURE proc(x NUMBER),
   MEMBER PROCEDURE proc2(x NUMBER)
);
```

### Example 14-21 Deprecation of a Member Function in an Object Type Specification

This example shows the deprecation of member function add2 in an object type specification.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of a Member Function in an Object Type Specification

```
TYPE objdata AS OBJECT(
  n1 NUMBER ,
  n2 NUMBER ,
  n3 NUMBER ,
  MEMBER FUNCTION add2 RETURN NUMBER ,
   PRAGMA DEPRECATE (add2),
  MEMBER FUNCTION add_all RETURN NUMBER
 );
```

### Example 14-22 Deprecation of Inherited Object Types

This example shows that a reference to a deprecated entity x declared in unit type15_basetype type body will cause the compiler to issue a warning.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation of Inherited Object Types

```
TYPE type15_basetype AS OBJECT
(
  x1 NUMBER,
  x  NUMBER,
  PRAGMA DEPRECATE (x),
  MEMBER PROCEDURE f0 ,
  PRAGMA DEPRECATE (f0),
  MEMBER PROCEDURE f1 ,
  PRAGMA DEPRECATE (f1),
  MEMBER PROCEDURE f2 ,
  PRAGMA DEPRECATE (f2),
  MEMBER PROCEDURE f3) NOT FINAL;


TYPE BODY type15_basetype AS
   MEMBER PROCEDURE f0
   IS
   BEGIN
     x := 1;
   END;
   MEMBER PROCEDURE f1
   IS
```

```
   BEGIN
     x := 1;
   END;

   MEMBER PROCEDURE f2
   IS
   BEGIN
     x := 1;
   END;

   MEMBER PROCEDURE f3
   IS
   BEGIN
     x := 1;
   END;
END;
```

References to the deprecated entities x, f0, and f2 in type15_basetype type body will cause the compiler to issue a warning.

```
TYPE type15_subtype UNDER type15_basetype (
  y NUMBER ,
  MEMBER PROCEDURE f1(z NUMBER),
  MEMBER PROCEDURE f1(z NUMBER , m1 NUMBER),
  PRAGMA DEPRECATE(f1),
  OVERRIDING MEMBER PROCEDURE f2
);


TYPE BODY type15_subtype AS
   MEMBER PROCEDURE f1(z NUMBER)
IS
BEGIN
  -- deprecation attribute inherited in derived type.
  x := 1;
  x1:= 2;
  SELF.f0;
END;

   MEMBER PROCEDURE f1(z  NUMBER ,
                       m1 NUMBER)
   IS
   BEGIN
     NULL;
   END;
   OVERRIDING MEMBER PROCEDURE f2
   IS
   BEGIN
     /* refer to deprecated f2 in supertype */
     (SELF AS type15_basetype).f2;
     /* No warning for a reference to a not deprecated data member in the
supertype */
     x1 := 1;
   END;
END;
```

References to deprecated entities x, f1, and f0 in unit type15_basetype will cause the compiler to issue a warning.

```
PROCEDURE test_types3
AS
  e type15_subtype ;
  d type15_basetype ;
BEGIN
  e := type15_subtype (1 ,1 ,1);
  d := type15_basetype (1, 1);
  d.x := 2; -- warning issued
  d.f1;     -- warning issued
  e.f1 (4); -- overloaded in derived type. no warning. not deprecated in the
derived type.
  e.f1 (1); -- no warning
  e.f0;     -- f0 is deprecated in base type. deprecation is inherited.
warning issued
            -- warning issued for deprecated x in d.x and e.x

  DBMS_OUTPUT.PUT_LINE(to_char(e.x) || to_char(' ') || to_char(d.x));

END;
```

### Example 14-23 Deprecation Only Applies to Top Level Subprogram

This examples shows that the `DEPRECATE` pragma may not be used to deprecate a nested procedure. The compiler issues a warning about the misuse of the pragma on the entity. The pragma has no effect.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Deprecation Only Applies to Top Level Subprogram

```
PROCEDURE foo
IS
   PROCEDURE inner_foo
   IS
   PRAGMA DEPRECATE (inner_foo, 'procedure inner_foo is deprecated');
   BEGIN
     DBMS_OUTPUT.PUT_LINE('Executing inner_foo');
   END;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Executing foo');
END;
```

### Example 14-24 Misplaced DEPRECATE Pragma

The `DEPRECATE` pragma must appear immediately after the declaration of the deprecated item. A warning about the misplaced pragma will be issued and the pragma will have no effect.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Misplaced DEPRECATE Pragma

```
PROCEDURE bar
IS
BEGIN
   PRAGMA DEPRECATE(bar);
   DBMS_OUTPUT.PUT_LINE('Executing bar.');
END;
```

**Example 14-25 Mismatch of the Element Name and the DEPRECATE Pragma Argument**

This example shows that if the argument for the pragma does not match the name in the declaration, the pragma is ignored and the compiler does not issue a warning.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at Mismatch of the Element Name and the DEPRECATE Pragma Argument

```
PACKAGE pkg13
AS
  PRAGMA DEPRECATE ('pkg13', 'Package pkg13 is deprecated, use pkg03');
  Y NUMBER;
END pkg13;
```

If an identifier is applied with a mismatched name, then the compiler issues a warning about the pragma being misplaced. The pragma has no effect.

```
CREATE PACKAGE pkg17
IS
  PRAGMA DEPRECATE ("pkg17");
END pkg17;
```

**Related Topics**

In this book:

- Pragmas
- PL/SQL Units and Compilation Parameters for more information about setting the `PLSQL_WARNINGS` compilation parameter

In other books:

- *Oracle Development Guide* for more information about deprecating packages, subprograms, and types

- *Oracle Database PL/SQL Packages and Types Reference* for more information about enabling the deprecation warnings using the `DBMS_WARNING.ADD_WARNING_SETTING_NUM` procedure
- Compile-Time Warnings for more information compilation warnings.

# DETERMINISTIC Clause

The deterministic option marks a function that returns predictable results and has no side effects.

Function-based indexes, virtual column definitions that use PL/SQL functions, and materialized views that have query-rewrite enabled require special function properties. The `DETERMINISTIC` clause asserts that a function has those properties.

The DETERMINISTIC option can appear in the following statements:

- Function Declaration and Definition
- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE TYPE BODY Statement

**Topics**

- Syntax
- Semantics
- Usage Notes
- Related Topics

**Syntax**

***deterministic_clause* ::=**


**Semantics**

***deterministic_clause***

**DETERMINISTIC**

A function is deterministic if the `DETERMINISTIC` clause appears in either a declaration or the definition of the function.

The `DETERMINISTIC` clause may appear at most once in a function declaration and at most once in a function definition.

A deterministic function must return the same value on two distinct invocations if the arguments provided to the two invocations are the same.

A `DETERMINISTIC` function may not have side effects.

A `DETERMINISTIC` function may not raise an unhandled exception.

If a function with a DETERMINISTIC clause violates any of these semantic rules, the results of its invocation, its value, and the effect on its invoker are all undefined.

**Usage Notes**

The DETERMINISTIC clause is an assertion that the function obeys the semantic rules. If the function does not, neither the compiler, SQL execution, or PL/SQL execution may diagnose the problem and wrong results may be silently produced.

You must specify this keyword if you intend to invoke the function in the expression of a function-based index, in a virtual column definition, or from the query of a materialized view that is marked REFRESH FAST or ENABLE QUERY REWRITE. When the database encounters a deterministic function, it tries to use previously calculated results when possible rather than reexecuting the function. If you change the function, then you must manually rebuild all dependent function-based indexes and materialized views.

Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects, because results might vary across invocations.

Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the return result of the function.

Specifying this clause for polymorphic table function is not allowed.

When the DETERMINISTIC option appears, the compiler may use the mark to improve the performance of the execution of the function.

It is good programming practice to make functions that fall into these categories DETERMINISTIC:

- Functions used in a WHERE, ORDER BY, or GROUP BY clause
- Functions that MAP or ORDER methods of a SQL type
- Functions that help determine whether or where a row appears in a result set

**Related Topics**

In other chapters:

- "Subprogram Side Effects"

In other books:

- CREATE INDEX statement in *Oracle Database SQL Language Reference*
- Oracle Database Data Warehousing Guide for information about materialized views
- *Oracle Database SQL Language Reference* for information about function-based indexes

# Element Specification

An element specification specifies each attribute of the ADT.

An element specification can appear in the following SQL statements :

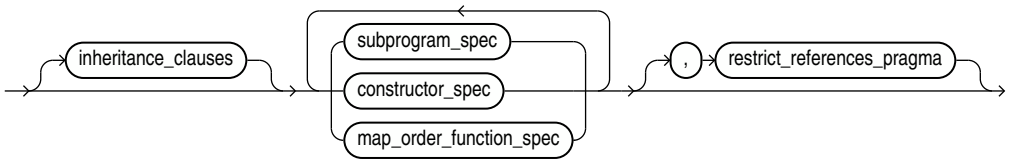
- ALTER TYPE Statement
- CREATE TYPE Statement

### Topics

- Syntax
- Semantics

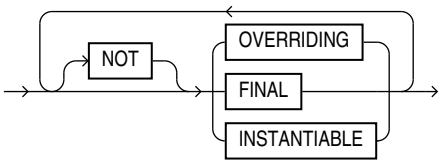
### Syntax

***element_spec* ::=**

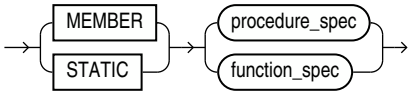



( *subprogram_spec* ::= , *constructor_spec* ::=, *map_order_function_spec* ::=, *restrict_references_pragma* ::=)

***inheritance_clauses* ::=**

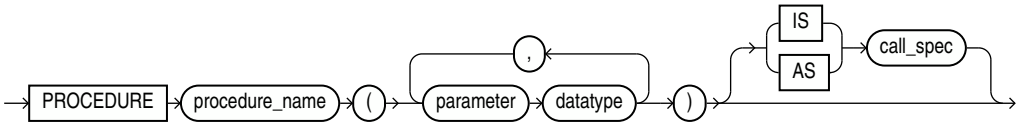



***subprogram_spec* ::=**



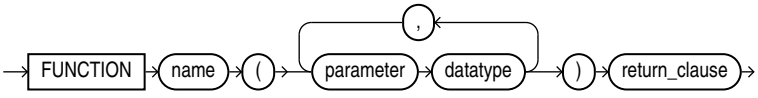

( *procedure_spec* ::=, *function_spec* ::= )

***procedure_spec* ::=**

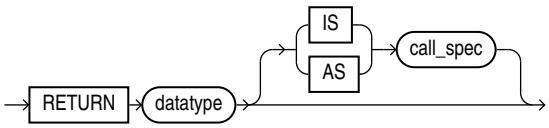



(*call_spec* ::=)

***function_spec* ::=**

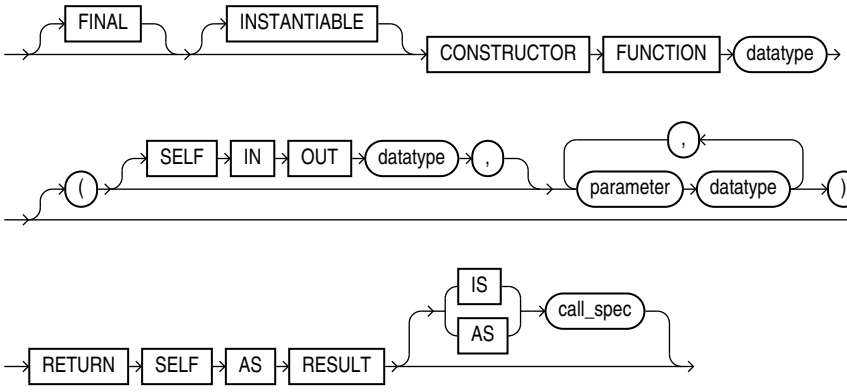


***return_clause* ::=**



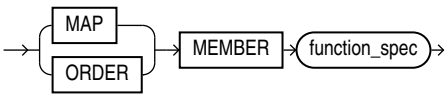

(*call_spec* ::=)

***constructor_spec* ::=**




(*call_spec* ::=)

***map_order_function_spec* ::=**




(*function_spec* ::=)

**Semantics**

***element_spec***

***inheritance_clauses***

Specifies the relationship between supertypes and subtypes.

**[NOT] OVERRIDING**

Specifies that this method overrides a `MEMBER` method defined in the supertype. This keyword is required if the method redefines a supertype method. **Default:** `NOT OVERRIDING`.

**[NOT] FINAL**

Specifies that this method cannot be overridden by any subtype of this type. **Default:** `NOT FINAL`.

**[NOT] INSTANTIABLE**

Specifies that the type does not provide an implementation for this method. **Default:** all methods are `INSTANTIABLE`.

**Restriction on NOT INSTANTIABLE**

If you specify `NOT INSTANTIABLE`, then you cannot specify `FINAL` or `STATIC`.

> **See Also:**
>
> *constructor_spec*

***subprogram_spec***

Specifies a subprogram to be referenced as an ADT attribute. For each such subprogram, you must specify a corresponding method body in the ADT body.

**Restriction on *subprogram_spec***

You cannot define a `STATIC` method on a subtype that redefines a `MEMBER` method in its supertype, or vice versa.

**MEMBER**

A subprogram associated with the ADT that is referenced as an attribute. Typically, you invoke `MEMBER` methods in a selfish style, such as *`object_expression.method()`*. This class of method has an implicit first argument referenced as `SELF` in the method body, which represents the object on which the method was invoked.

> **See Also:**
>
> "Example 15-35"

**STATIC**

A subprogram associated with the ADT. Unlike `MEMBER` methods, `STATIC` methods do not have any implicit parameters. You cannot reference `SELF` in their body. They are typically invoked as *`type_name.method()`*.

**Restrictions on STATIC**

- You cannot map a `MEMBER` method in a Java class to a `STATIC` method in a SQLJ object type.
- For both `MEMBER` and `STATIC` methods, you must specify a corresponding method body in the type body for each procedure or function specification.

> **See Also:**
>
> "Example 15-36"

***procedure_spec* or *function_spec***

Specifies the parameters and data types of the procedure or function. If this subprogram does not include the declaration of the procedure or function, then you must issue a corresponding `CREATE TYPE BODY` statement.

**Restriction on *procedure_spec* or *function_spec***

If you are creating a subtype, then the name of the procedure or function cannot be the same as the name of any attribute, whether inherited or not, declared in the supertype chain.

***return_clause***

The first form of the *return_clause* is valid only for a function. The syntax shown is an abbreviated form.

> **See Also:**
>
> "Collection Method Invocation" for information about method invocation and methods

***constructor_spec***

Creates a user-defined constructor, which is a function that returns an initialized instance of an ADT. You can declare multiple constructors for a single ADT, if the parameters of each constructor differ in number, order, or data type.

- User-defined constructor functions are always `FINAL` and `INSTANTIABLE`, so these keywords are optional.
- The parameter-passing mode of user-defined constructors is always `SELF IN OUT`. Therefore you need not specify this clause unless you want to do so for clarity.
- `RETURN SELF AS RESULT` specifies that the runtime type of the value returned by the constructor is runtime type of the `SELF` argument.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information about and examples of user-defined constructors and "Example 15-34"

***map_order_function_spec***

You can declare either one `MAP` method or one `ORDER` method in a type specification, regardless of how many `MEMBER` or `STATIC` methods you declare. If you declare either method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. You must not specify a comparison method to determine the equality of two ADTs.

You cannot declare either `MAP` or `ORDER` methods for subtypes. However, a subtype can override a `MAP` method if the supertype defines a `NOT FINAL MAP` method. A subtype cannot override an `ORDER` method at all.

You can specify either `MAP` or `ORDER` when mapping a Java class to a SQL type. However, the `MAP` or `ORDER` methods must map to `MEMBER` functions in the Java class.

If neither a `MAP` nor an `ORDER` method is specified, then only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method must be specified to determine the equality of two ADTs.

Use `MAP` if you are performing extensive sorting or hash join operations on object instances. `MAP` is applied once to map the objects to scalar values, and then the database uses the scalars during sorting and merging. A `MAP` method is more efficient than an `ORDER` method, which must invoke the method for each object comparison. You must use a `MAP` method for hash joins. You cannot use an `ORDER` method because the hash mechanism hashes on the object value.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information about object value comparisons

**MAP MEMBER**

Specifies a `MAP` member function (`MAP` method) that returns the relative position of a given instance in the ordering of all instances of the object. A `MAP` method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the `MAP` method is `NULL`, then the `MAP` method returns `NULL` and the method is not invoked.

An object specification can contain only one `MAP` method, which must be a function. The result type must be a predefined SQL scalar type, and the `MAP` method can have no arguments other than the implicit `SELF` argument.

> **Note:**
>
> If *`type_name`* is to be referenced in queries containing sorts (through an `ORDER BY`, `GROUP BY`, `DISTINCT`, or `UNION` clause) or containing joins, and you want those queries to be parallelized, then you must specify a `MAP` member function.

A subtype cannot define a new `MAP` method, but it can override an inherited `MAP` method.

**ORDER MEMBER**

Specifies an `ORDER` member function (`ORDER` method) that takes an instance of an object as an explicit argument and the implicit `SELF` argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit `SELF` argument is less than, equal to, or greater than the explicit argument.

If either argument to the `ORDER` method is `NULL`, then the `ORDER` method returns `NULL` and the method is not invoked.

When instances of the same ADT definition are compared in an `ORDER BY` clause, the `ORDER` method function is invoked.

An object specification can contain only one `ORDER` method, which must be a function having the return type `NUMBER`.

A subtype can neither define nor override an `ORDER` method.

**Restriction on *map_order_function_spec***

You cannot add an `ORDER` method to a subtype.

***restrict_references_pragma***

Deprecated clause, described in "RESTRICT_REFERENCES Pragma".

**Restriction on *restrict_references_pragma***

This clause is not valid when dropping a method.

# EXCEPTION_INIT Pragma

The `EXCEPTION_INIT` pragma associates a user-defined exception name with an error code.

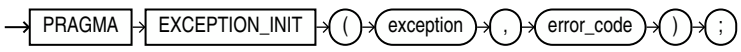
The `EXCEPTION_INIT` pragma can appear only in the same declarative part as its associated exception, anywhere after the exception declaration.

**Topics**

- Syntax
- Semantics
- Usage Notes
- Examples
- Related Topics

**Syntax**

***exception_init_pragma* ::=**




**Semantics**

***exception_init_pragma***

***exception***

Name of a previously declared user-defined exception.

***error_code***

Error code to be associated with *exception*. *error_code* can be either 100 (the numeric code for "no data found" that "SQLCODE Function" returns) or any negative integer greater than -1000000 except -1403 (another numeric code for "no data found").

> **Note:**
>
> NO_DATA_FOUND is a predefined exception.

If two EXCEPTION_INIT pragmas assign different error codes to the same user-defined exception, then the later pragma overrides the earlier pragma.

### Usage Notes

The EXCEPTION_INIT pragma should only be used to associate an exception with an error number that is already meaningfully defined by Oracle. Note that any error number may be used by Oracle in the future, which can create conflicts with unrelated application use of that number.

Negative integers greater than -65536 are only partially supported. Currently, if left unhandled beyond the current layer of entry into PL/SQL, the exception is converted to ORA-6515 and the original exception is not recognized in the outer PL/SQL layer or client program.

Application-declared exceptions that are only raised and caught locally within a layer of entry into PL/SQL do not need the EXCEPTION_INIT pragma. The RAISE_APPLICATION_ERROR procedure and associated -20000 to -20999 range of error numbers should be used by application-declared exceptions that are intended to be recognizable in outer PL/SQL layers or in the client program.

### Examples

- Example 12-5, "Naming Internally Defined Exception"
- Example 12-13, "Raising User-Defined Exception with RAISE_APPLICATION_ERROR"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"

### Related Topics

In this chapter:

- "Exception Declaration"
- "Exception Handler"
- "SQLCODE Function"
- "SQLERRM Function"

In other chapters:

- "Internally Defined Exceptions"
- "RAISE_APPLICATION_ERROR Procedure"

# Exception Declaration

An exception declaration declares the name of a user-defined exception.

You can use the `EXCEPTION_INIT` pragma to assign this name to an internally defined exception.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***exception_declaration* ::=**


**Semantics**

***exception_declaration***

***exception***

Name of the exception that you are declaring.

**Restriction on *exception***

You can use *`exception`* only in an `EXCEPTION_INIT` pragma, `RAISE` statement, `RAISE_APPLICATION_ERROR` invocation, or exception handler.

> **Caution:**
>
> Oracle recommends against using a predefined exception name for *`exception`*. For details, see "Redeclared Predefined Exceptions". For a list of predefined exception names, see Table 12-3.

**Examples**

- Example 12-5, "Naming Internally Defined Exception"
- Example 12-9, "Redeclared Predefined Identifier"
- Example 12-10, "Declaring, Raising, and Handling User-Defined Exception"

**Related Topics**

In this chapter:

- "EXCEPTION_INIT Pragma"
- "Exception Handler"
- "RAISE Statement"

In other chapters:

- "Internally Defined Exceptions"
- "User-Defined Exceptions"

# Exception Handler

An exception handler processes a raised exception.

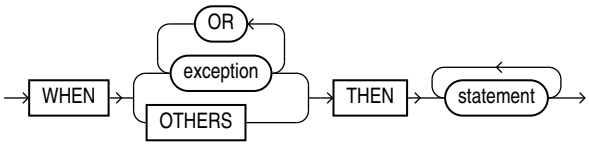
Exception handlers appear in the exception-handling parts of anonymous blocks, subprograms, triggers, and packages.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***exception_handler* ::=**




(*statement* ::=)

**Semantics**

***exception_handler***

***exception***

Name of either a predefined exception (see Table 12-3) or a user-defined exception (see "Exception Declaration").

If PL/SQL raises a specified exception, then the associated statements run.

**OTHERS**

Specifies all exceptions not explicitly specified in the exception handling part of the block. If PL/SQL raises such an exception, then the associated statements run.

> **Note:**
>
> Oracle recommends that the last statement in the `OTHERS` exception handler be either `RAISE` or an invocation of a subroutine marked with pragma `SUPPRESSES_WARNING_6009`.
>
> If you do not follow this practice, and PL/SQL warnings are enabled, you get `PLW-06009`.

In the exception handling part of a block, the `WHEN OTHERS` exception handler is optional. It can appear only once, as the last exception handler in the exception handling part of the block.

**Examples**

- Example 12-3, "Single Exception Handler for Multiple Exceptions"
- Example 12-4, "Locator Variables for Statements that Share Exception Handler"
- Example 12-6, "Anonymous Block Handles ZERO_DIVIDE"
- Example 12-7, "Anonymous Block Avoids ZERO_DIVIDE"
- Example 12-10, "Declaring, Raising, and Handling User-Defined Exception"
- Example 12-14, "Exception that Propagates Beyond Scope is Handled"
- Example 12-24, "Exception Handler Runs and Execution Ends"
- Example 12-25, "Exception Handler Runs and Execution Continues"
- Example 13-12, "Handling FORALL Exceptions Immediately"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"

**Related Topics**

In this chapter:

- "Block"
- "EXCEPTION_INIT Pragma"
- "Exception Declaration"
- "RAISE Statement"
- "SQLCODE Function"
- "SQLERRM Function"

In other chapters:

- "Overview of Exception Handling"
- "Continuing Execution After Handling Exceptions"
- "Retrying Transactions After Handling Exceptions"
- "CREATE PACKAGE BODY Statement"
- "CREATE TRIGGER Statement"

# EXECUTE IMMEDIATE Statement

The `EXECUTE IMMEDIATE` statement builds and runs a dynamic SQL statement in a single operation.

Native dynamic SQL uses the `EXECUTE IMMEDIATE` statement to process most dynamic SQL statements.

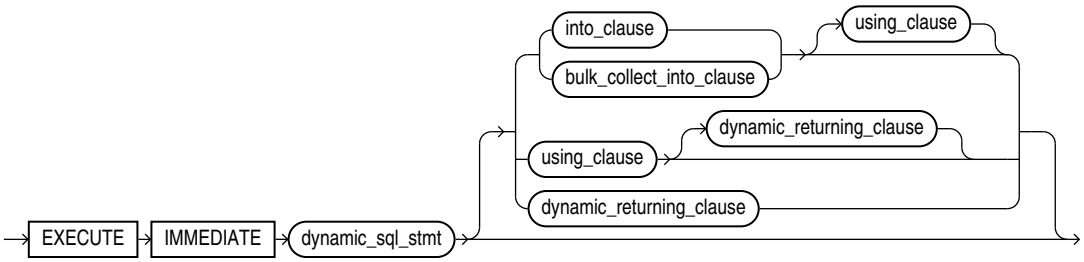
> **Caution:**
>
> When using dynamic SQL, beware of SQL injection, a security risk. For more information about SQL injection, see "SQL Injection".

**Topics**


- Syntax
- Semantics
- Examples
- Related Topics

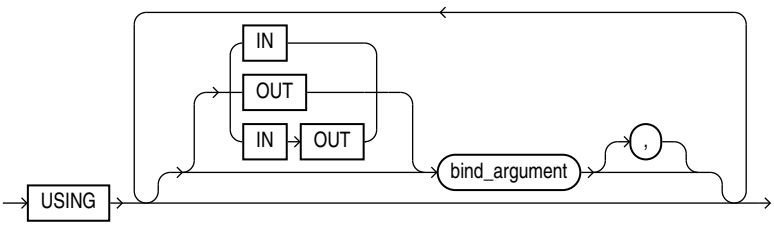

**Syntax**

***execute_immediate_statement* ::=**




(*bulk_collect_into_clause* ::=, *dynamic_returning_clause* ::=, *into_clause* ::=)

***using_clause* ::=**



### Semantics

***execute_immediate_statement***

***dynamic_sql_stmt***

String literal, string variable, or string expression that represents a SQL statement. Its type must be either `CHAR`, `VARCHAR2`, or `CLOB`.

> **Note:**
>
> If *dynamic_sql_statement* is a `SELECT` statement, and you omit both *into_clause* and *bulk_collect_into_clause*, then *execute_immediate_statement* never runs.
>
> For example, this statement never increments the sequence:
>
> ```
> EXECUTE IMMEDIATE 'SELECT S.NEXTVAL FROM DUAL'
> ```

***into_clause***

Specifies the variables or record in which to store the column values that the statement returns. For more information about this clause, see "RETURNING INTO Clause".

**Restriction on *into_clause***

Use if and only if *dynamic_sql_stmt* returns a single row.

***bulk_collect_into_clause***

Specifies one or more collections in which to store the rows that the statement returns. For more information about this clause, see "RETURNING INTO Clause".

**Restriction on *bulk_collect_into_clause***

Use if and only if *dynamic_sql_stmt* can return multiple rows.

***dynamic_returning_clause***

Returns the column values of the rows affected by the dynamic SQL statement, in either individual variables or records. For more information about this clause, see "RETURNING INTO Clause".

**Restriction on *dynamic_returning_clause***

Use if and only if *dynamic_sql_stmt* has a `RETURNING INTO` clause.

***using_clause***

Specifies bind variables, using positional notation.

> **Note:**
>
> If you repeat placeholder names in *dynamic_sql_statement*, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement. For details, see "Repeated Placeholder Names in Dynamic SQL Statements."

**Restrictions on *using_clause***

- Use if and only if *dynamic_sql_stmt* includes placeholders for bind variables.
- If *dynamic_sql_stmt* has a `RETURNING INTO` clause (*static_returning_clause*), then *using_clause* can contain only `IN` bind variables. The bind variables in the `RETURNING INTO` clause are `OUT` bind variables by definition.

**IN, OUT, IN OUT**

Parameter modes of bind variables. An `IN` bind variable passes its value to *dynamic_sql_stmt*. An `OUT` bind variable stores a value that *dynamic_sql_stmt* returns. An `IN OUT` bind variable passes its initial value to *dynamic_sql_stmt* and stores a value that *dynamic_sql_stmt* returns. **Default:** `IN`.

For DML a statement with a `RETURNING` clause, you can place `OUT` bind variables in the `RETURNING INTO` clause without specifying the parameter mode, which is always `OUT`.

***bind_argument***

An expression whose value replaces its corresponding placeholder in *dynamic_sql_stmt* at run time.

Every placeholder in *dynamic_sql_stmt* must be associated with a *bind_argument* in the `USING` clause or `RETURNING INTO` clause (or both) or with a define variable in the `INTO` clause.

You can run *dynamic_sql_stmt* repeatedly using different values for the bind variables. You incur some overhead, because `EXECUTE IMMEDIATE` prepares the dynamic string before every execution.

> **Note:**
>
> Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.

**Restrictions on *bind_argument***

- *bind_argument* cannot be an associative array indexed by string.
- *bind_argument* cannot be the reserved word `NULL`.

  To pass the value `NULL` to the dynamic SQL statement, use an uninitialized variable where you want to use `NULL`, as in Example 8-7.

**Examples**

- Example 8-1, "Invoking Subprogram from Dynamic PL/SQL Block"

- Example 8-7, "Uninitialized Variable Represents NULL in USING Clause"
- Example 8-10, "Repeated Placeholder Names in Dynamic PL/SQL Block"

**Related Topics**

In this chapter:

- "RETURNING INTO Clause"

In other chapters:

- "EXECUTE IMMEDIATE Statement"
- "DBMS_SQL Package"

# EXIT Statement

The `EXIT` statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the end of either the current loop or an enclosing labeled loop.

The `EXIT WHEN` statement exits the current iteration of a loop when the condition in its `WHEN` clause is true, and transfers control to the end of either the current loop or an enclosing labeled loop.

Each time control reaches the `EXIT WHEN` statement, the condition in its `WHEN` clause is evaluated. If the condition is not true, the `EXIT WHEN` statement does nothing. To prevent an infinite loop, a statement inside the loop must make the condition true.

**Restriction on EXIT Statement**

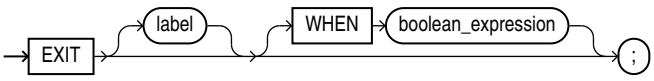
An `EXIT` statement must be inside a `LOOP` statement.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***exit_statement* ::=**




(*boolean_expression* ::=)

**Semantics**

***exit_statement***

***label***

Name that identifies either the current loop or an enclosing loop.

Without *label*, the EXIT statement transfers control to the end of the current loop. With *label*, the EXIT statement transfers control to the end of the loop that *label* identifies.

**WHEN *boolean_expression***

Without this clause, the EXIT statement exits the current iteration of the loop unconditionally. With this clause, the EXIT statement exits the current iteration of the loop if and only if the value of *boolean_expression* is TRUE.

### Examples

#### Example 14-26 Basic LOOP Statement with EXIT Statement

In this example, the EXIT statement inside the basic LOOP statement transfers control unconditionally to the end of the current loop.

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

#### Example 14-27 Basic LOOP Statement with EXIT WHEN Statement

In this example, the EXIT WHEN statement inside the basic LOOP statement transfers control to the end of the current loop when x is greater than 3.

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;  -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
  DBMS_OUTPUT.PUT_LINE('After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

**Related Topics**

- "Basic LOOP Statement"
- "CONTINUE Statement"

# Explicit Cursor Declaration and Definition

An **explicit cursor** is a named pointer to a private SQL area that stores information for processing a specific query or DML statement—typically, one that returns or affects multiple rows.

You can use an explicit cursor to retrieve the rows of a result set one at a time.

Before using an explicit cursor, you must declare and define it. You can either declare it first (with ***cursor_declaration***) and then define it later in the same block, subprogram, or package (with ***cursor_definition***) or declare and define it at the same time (with ***cursor_definition***).

An explicit cursor declaration and definition are also called a **cursor specification** and **cursor body**, respectively.

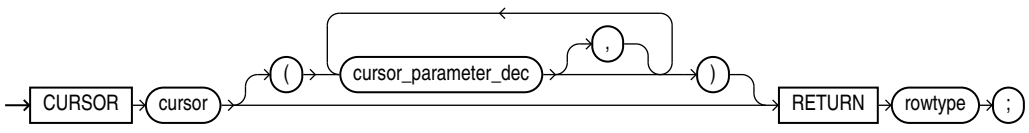
> **Note:**
>
> An explicit cursor declared in a package specification is affected by the `AUTHID` clause of the package. For more information, see "CREATE PACKAGE Statement".

**Topics**

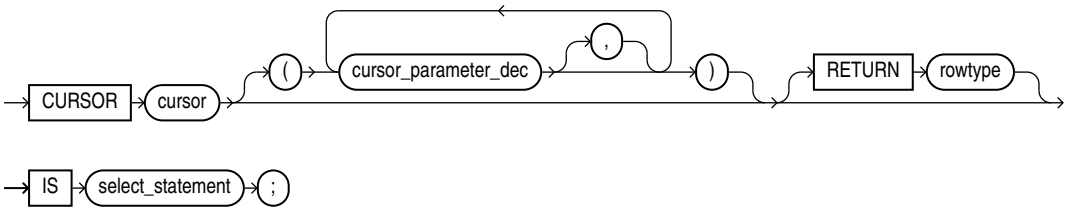
- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

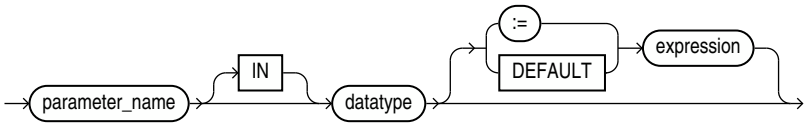
***cursor_declaration* ::=**



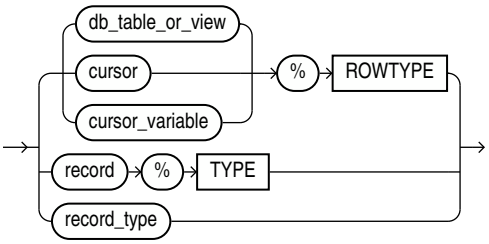
***cursor_definition* ::=**




***cursor_parameter_dec* ::=**




***rowtype* ::=**




### Semantics

#### *cursor_declaration*

#### *cursor*

Name of the explicit cursor that you are declaring now and will define later in the same block, subprogram, or package. *cursor* can be any identifier except the reserved word SQL. Oracle recommends against giving a cursor the same name as a database table.

Explicit cursor names follow the same scoping rules as variables (see "Scope and Visibility of Identifiers").

#### *cursor_definition*

Either defines an explicit cursor that was declared earlier or both declares and defines an explicit cursor.

#### *cursor*

Either the name of the explicit cursor that you previously declared and are now defining or the name of the explicit cursor that you are both declaring and defining. *cursor* can be any identifier except the reserved word SQL. Oracle recommends against giving a cursor the same name as a database table.

***select_statement***

A SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement). If the cursor has formal parameters, each parameter must appear in *`select_statement`*. The *`select_statement`* can also reference other PL/SQL variables in its scope.

**Restriction on *select_statement***

This *`select_statement`* cannot have a `WITH` clause.

> **See:**
>
> *Oracle Database SQL Language Reference* for `SELECT` statement syntax

***cursor_parameter_dec***

A cursor parameter declaration.

***parameter***

The name of the formal cursor parameter that you are declaring. This name can appear anywhere in *`select_statement`* that a constant can appear.

**IN**

Whether or not you specify `IN`, a formal cursor parameter has the characteristics of an `IN` subprogram parameter, which are summarized in Table 9-1. When the cursor opens, the value of the formal parameter is that of either its actual parameter or default value.

***datatype***

The data type of the parameter.

**Restriction on *datatype***

This *`datatype`* cannot have constraints (for example, `NOT NULL`, or precision and scale for a number, or length for a string).

***expression***

Specifies the default value for the formal cursor parameter. The data types of *`expression`* and the formal cursor parameter must be compatible.

If an `OPEN` statement does not specify an actual parameter for the formal cursor parameter, then the statement evaluates *`expression`* and assigns its value to the formal cursor parameter.

If an `OPEN` statement does specify an actual parameter for the formal cursor parameter, then the statement assigns the value of the actual parameter to the formal cursor parameter and does not evaluate *`expression`*.

***rowtype***

Data type of the row that the cursor returns. The columns of this row must match the columns of the row that *`select_statement`* returns.

***db_table_or_view***

Name of a database table or view, which must be accessible when the declaration is elaborated.

***cursor***

Name of a previously declared explicit cursor.

***cursor_variable***

Name of a previously declared cursor variable.

***record***

Name of a previously declared record variable.

***record_type***

Name of a user-defined type that was defined with the data type specifier `RECORD`.

**Examples**

- Example 7-5, "Explicit Cursor Declaration and Definition"
- Example 7-8, "Variable in Explicit Cursor Query—No Result Set Change"
- Example 7-9, "Variable in Explicit Cursor Query—Result Set Change"
- Example 7-10, "Explicit Cursor with Virtual Column that Needs Alias"
- Example 7-11, "Explicit Cursor that Accepts Parameters"
- Example 7-12, "Cursor Parameters with Default Values"
- Example 7-13, "Adding Formal Parameter to Existing Cursor"
- Example 7-22, "Subquery in FROM Clause of Parent Query"
- Example 7-23, "Correlated Subquery"
- Example 7-35, "CURSOR Expression"
- Example 7-41, "FETCH with FOR UPDATE Cursor After COMMIT Statement"

**Related Topics**

In this chapter:

- "CLOSE Statement"
- "Cursor FOR LOOP Statement"
- "Cursor Variable Declaration"
- "FETCH Statement"
- "Named Cursor Attribute"
- "OPEN Statement"
- "%ROWTYPE Attribute"
- "%TYPE Attribute"

In other chapters:

- "Explicit Cursors"
- "Processing Query Result Sets"
- "SELECT FOR UPDATE and FOR UPDATE Cursors"

# Expression

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function invocations, and placeholders) and operators.

The simplest expression is a single variable.

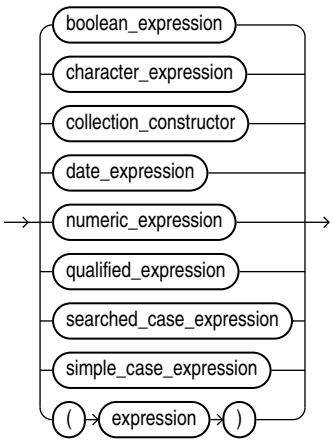
The PL/SQL compiler determines the data type of an expression from the types of the operands and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results.

**Topics**


- Syntax
- Semantics
- Examples
- Related Topics

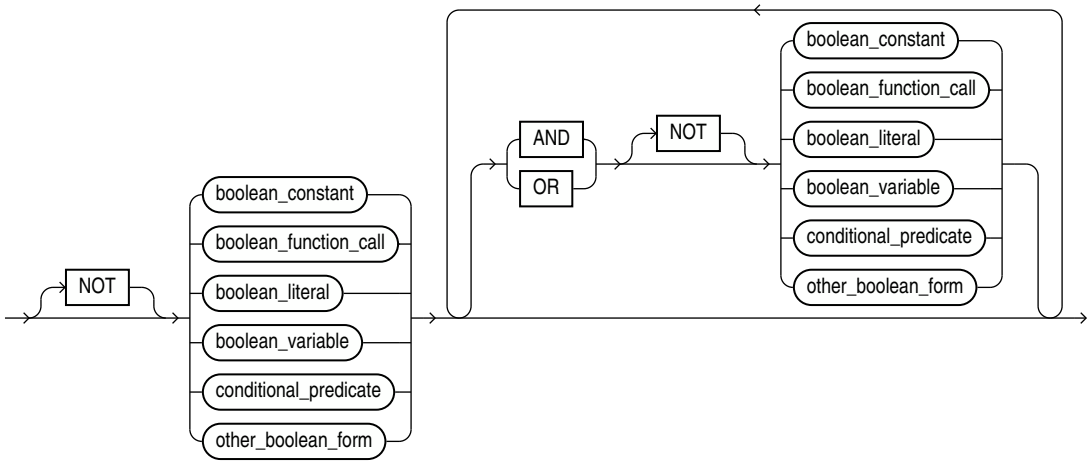

**Syntax**

***expression* ::=**



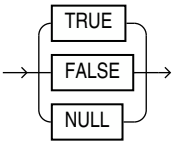

(*boolean_expression* ::=, *character_expression* ::=, *collection_constructor* ::=, *date_expression* ::=, *numeric_expression* ::=, *qualified_expression* ::= , *searched_case_expression* ::=, *simple_case_expression* ::=)

***boolean_expression* ::=**

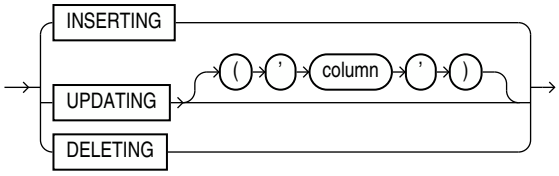



(*function_call* ::=)

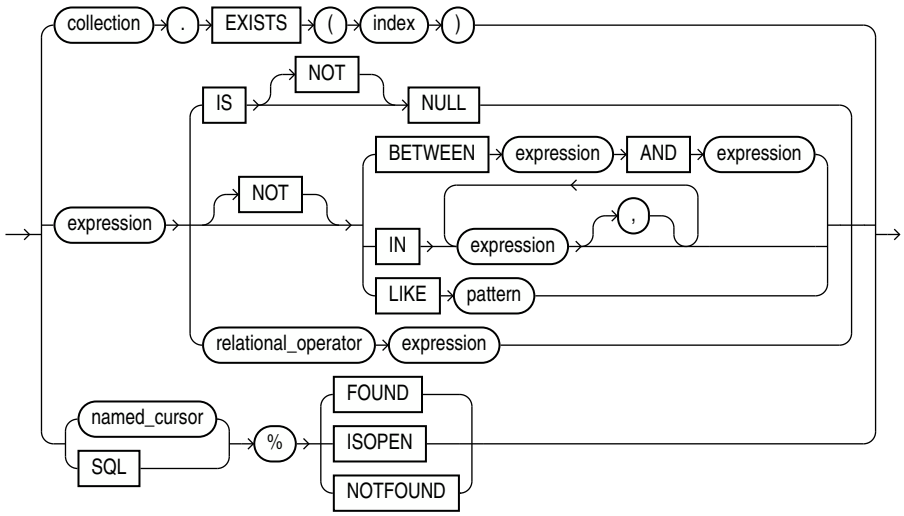
***boolean_literal* ::=**




***conditional_predicate* ::=**

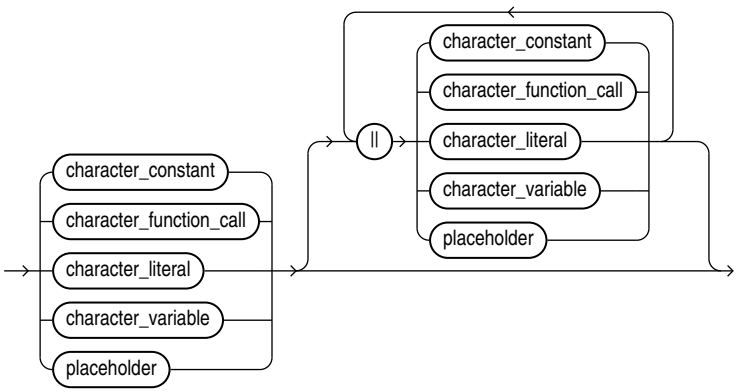


***other_boolean_form* ::=**



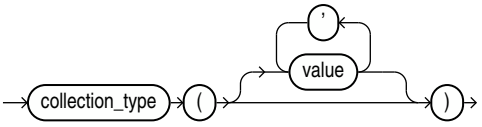

(*expression* ::=, *named_cursor* ::=)

***character_expression* ::=**

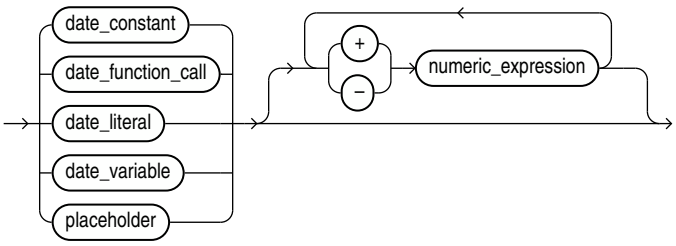



(*function_call* ::=, *placeholder* ::=)

***collection_constructor* ::=**

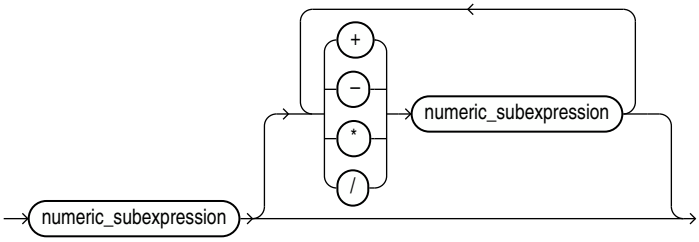


***date_expression* ::=**

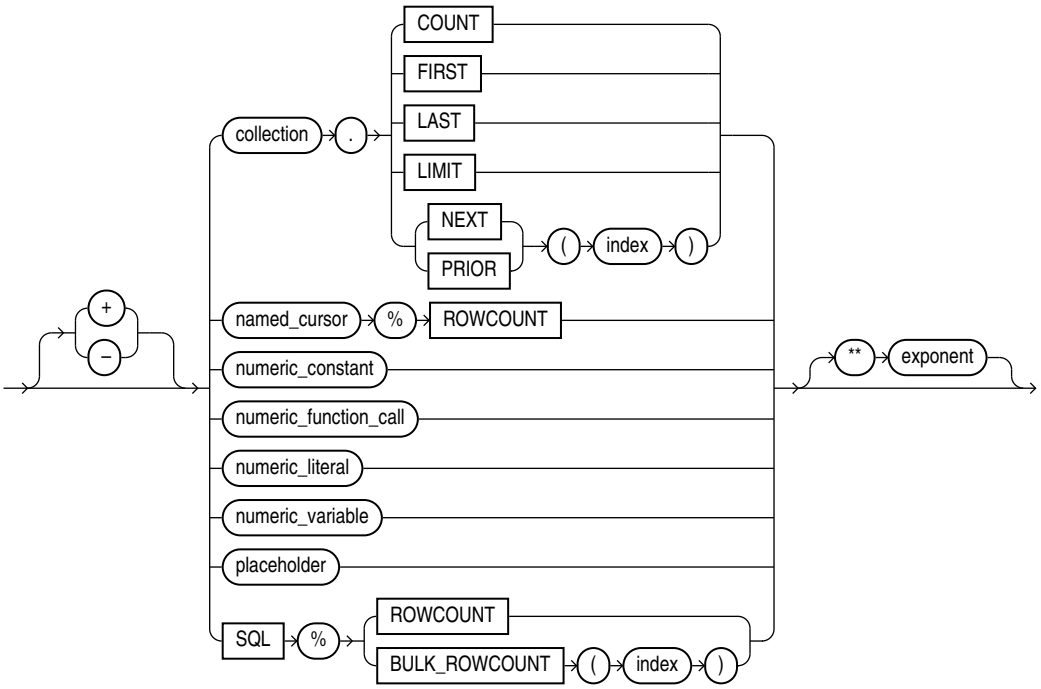



(*function_call* ::=, *placeholder* ::=)

***numeric_expression* ::=**

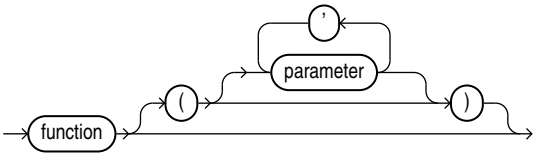



***numeric_subexpression* ::=**

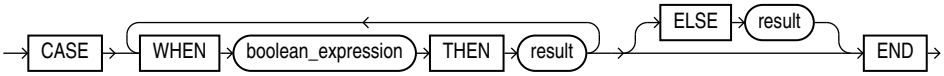



(*function_call* ::=, *named_cursor* ::=, *placeholder* ::=)

***function_call* ::=**

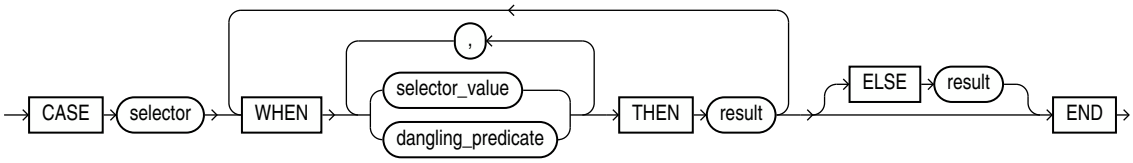



***searched_case_expression* ::=**




(*boolean_expression* ::=)

***simple_case_expression* ::=**




**Semantics**

***boolean_expression***

Expression whose value is `TRUE`, `FALSE`, or `NULL`. For more information, see "BOOLEAN Expressions".

**NOT, AND, OR**

See "Logical Operators".

***boolean_constant***

Name of a constant of type `BOOLEAN`.

***boolean_function_call***

Invocation of a previously defined function that returns a `BOOLEAN` value. For more semantic information, see "*function_call*".

***boolean_variable***

Name of a variable of type `BOOLEAN`.

***conditional_predicate***

See "Conditional Predicates for Detecting Triggering DML Statement".

***other_boolean_form***

***collection***

Name of a collection variable.

**EXISTS**

Collection method (function) that returns `TRUE` if the *`index`*th element of *`collection`* exists and `FALSE` otherwise. For more information, see "EXISTS Collection Method".

**Restriction on EXISTS**

You cannot use `EXISTS` if *`collection`* is an associative array.

***index***

Numeric expression whose data type either is `PLS_INTEGER` or can be implicitly converted to `PLS_INTEGER` (for information about the latter, see "Predefined PLS_INTEGER Subtypes").

**IS [NOT] NULL**

See "IS [NOT] NULL Operator".

**BETWEEN *expression* AND *expression***

See "BETWEEN Operator".

**IN *expression* [, *expression* ]...**

See "IN Operator".

**LIKE *pattern***

See "LIKE Operator".

***relational_operator***

See "Relational Operators".

**SQL**

Implicit cursor associated with the most recently run `SELECT` or DML statement. For more information, see "Implicit Cursors".

**%FOUND, %ISOPEN, %NOTFOUND**

Cursor attributes explained in "Implicit Cursor Attribute" and "Named Cursor Attribute".

***character_expression***

Expression whose value has a character data type (that is, a data type in the `CHAR` family, described in "CHAR Data Type Family").

***character_constant***

Name of a constant that has a character data type.

***character_function_call***

Invocation of a previously defined function that returns a value that either has a character data type or can be implicitly converted to a character data type. For more semantic information, see "*function_call*".

***character_literal***

Literal of a character data type.

***character_variable***

Name of a variable that has a character data type.

||

Concatenation operator, which appends one string operand to another. For more information, see "Concatenation Operator".

***collection_constructor***

Constructs a collection of the specified type with elements that have the specified values.

For more information, see "Collection Constructors".

***collection_type***

Name of a previously declared nested table type or `VARRAY` type (not an associative array type).

***value***

Valid value for an element of a collection of *`collection_type`*.

If *`collection_type`* is a varray type, then it has a maximum size, which the number of values cannot exceed. If *`collection_type`* is a nested table type, then it has no maximum size.

If you specify no values, then the constructed collection is empty but not null (for the difference between *empty* and *null*, see "Collection Types").

***date_expression***

Expression whose value has a date data type (that is, a data type in the `DATE` family, described in "DATE Data Type Family").

***date_constant***

Name of a constant that has a date data type.

***date_function_call***

Invocation of a previously defined function that returns a value that either has a date data type or can be implicitly converted to a date data type. For more semantic information, see "*function_call*".

***date_literal***

Literal whose value either has a date data type or can be implicitly converted to a date data type.

***date_variable***

Name of a variable that has a date data type.

**+, -**

Addition and subtraction operators.

***numeric_expression***

Expression whose value has a date numeric type (that is, a data type in the `DATE` family, described in "NUMBER Data Type Family").

**+, -, /, *, ****

Addition, subtraction, division, multiplication, and exponentiation operators.

***numeric_subexpression***

***collection***

Name of a collection variable.

**COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR**

Collection methods explained in "Collection Method Invocation".

***named_cursor*%ROWCOUNT**

See "Named Cursor Attribute".

***numeric_constant***

Name of a constant that has a numeric data type.

***numeric_function_call***

Invocation of a previously defined function that returns a value that either has a numeric data type or can be implicitly converted to a numeric data type. For more semantic information, see "*function_call*".

***numeric_literal***

Literal of a numeric data type.

***numeric_variable***

Name of variable that has a numeric data type.

**SQL%ROWCOUNT**

Cursor attribute explained in "Implicit Cursor Attribute".

**SQL%BULK_ROWCOUNT]**

Cursor attribute explained in "SQL%BULK_ROWCOUNT".

***exponent***

Numeric expression.

***function_call***

***function***

Name of a previously defined function.

***parameter* [, *parameter* ]...**

List of actual parameters for the function being called. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter. The mode of the formal parameter determines what the actual parameter can be:

| Formal Parameter Mode | Actual Parameter |
|---|---|
| IN | Constant, initialized variable, literal, or expression |
| OUT | Variable whose data type is not defined as NOT NULL |
| IN OUT | Variable (typically, it is a string buffer or numeric accumulator) |

If the function specifies a default value for a parameter, you can omit that parameter from the parameter list. If the function has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

> **See Also:**
>
> "Positional, Named, and Mixed Notation for Actual Parameters"

***searched_case_expression***

**WHEN *boolean_expression* THEN *result***

The *boolean_expression*s are evaluated sequentially. If a *boolean_expression* has the value TRUE, then the *result* associated with that *boolean_expression* is returned. Subsequent *boolean_expression*s are not evaluated.

**ELSE *result***

The *result* is returned if and only if no *boolean_expression* has the value TRUE.

If you omit the ELSE clause, the searched case expression returns NULL.

> **See Also:**
>
> "Searched CASE Statement"

***simple_case_expression***

***selector***

An expression of any PL/SQL type except BLOB, BFILE, or a user-defined type. The *selector* is evaluated once.

**WHEN *selector_value* THEN *result***

The *selector_values* are evaluated sequentially. If a *selector_value* matches the value of *selector*, then the *result* associated with that *selector_value* is returned. Subsequent *selector_values* are not evaluated.

A *selector_value* can be of any PL/SQL type except BLOB, BFILE, an ADT, a PL/SQL record, an associative array, a varray, or a nested table.

**ELSE *result***

The *result* is returned if and only if no *selector_value* has the same value as *selector*.

If you omit the `ELSE` clause, the simple case expression returns `NULL`.

> **Note:**
>
> If you specify the literal `NULL` for every *`result`* (including the *`result`* in the `ELSE` clause), then error PLS-00617 occurs.

> **See Also:**
>
> "Simple CASE Statement"

**Examples**

- Example 3-28, "Concatenation Operator Examples"
- Example 3-30, "Controlling Evaluation Order with Parentheses"
- Example 3-31, "Expression with Nested Parentheses"
- Example 3-32, "Improving Readability with Parentheses"
- Example 3-33, "Operator Precedence"
- Example 3-43, "Relational Operators in Expressions"
- Example 3-44, "LIKE Operator in Expression"
- Example 3-46, "BETWEEN Operator in Expressions"
- Example 3-47, "IN Operator in Expressions"
- Example 3-50, "Simple CASE Expression"
- Example 3-54, "Searched CASE Expression"
- Example 10-1, "Trigger Uses Conditional Predicates to Detect Triggering Statement"

**Related Topics**

In this chapter:

- "Collection Method Invocation"
- "Constant Declaration"
- Qualified Expression
- "Scalar Variable Declaration"

In other chapters:

- "Qualified Expressions Overview" for more information and examples
- "Literals"
- "Expressions"
- "Operator Precedence"
- "PL/SQL Data Types"
- "Subprogram Parameters"

# FETCH Statement

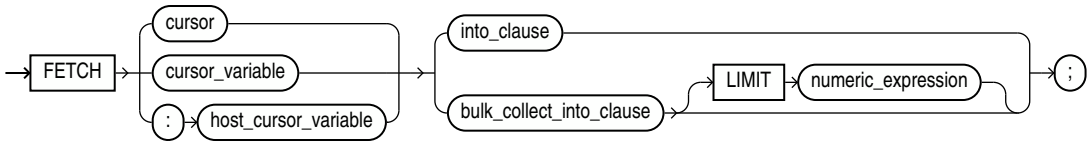
The `FETCH` statement retrieves rows of data from the result set of a multiple-row query—one row at a time, several rows at a time, or all rows at once—and stores the data in variables, records, or collections.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***fetch_statement* ::=**




(*bulk_collect_into_clause* ::=, *into_clause* ::=, *numeric_expression* ::=)

### Semantics

***fetch_statement***

***cursor***

Name of an open explicit cursor. To open an explicit cursor, use the "OPEN Statement".

If you try to fetch from an explicit cursor before opening it or after closing it, PL/SQL raises the predefined exception `INVALID_CURSOR`.

***cursor_variable***

Name of an open cursor variable. To open a cursor variable, use the "OPEN FOR Statement". The cursor variable can be a formal subprogram parameter (see "Cursor Variables as Subprogram Parameters").

If you try to fetch from a cursor variable before opening it or after closing it, PL/SQL raises the predefined exception `INVALID_CURSOR`.

***:host_cursor_variable***

Name of a cursor variable declared in a PL/SQL host environment, passed to PL/SQL as a bind variable, and then opened. To open a host cursor variable, use the "OPEN FOR Statement". Do not put space between the colon (:) and *`host_cursor_variable`*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

***into_clause***

To have the `FETCH` statement retrieve one row at a time, use this clause to specify the variables or record in which to store the column values of a row that the cursor returns. For more information about *`into_clause`*, see "*into_clause* ::=".

***bulk_collect_into_clause* [ LIMIT *numeric_expression* ]**

Use *`bulk_collect_into_clause`* to specify one or more collections in which to store the rows that the `FETCH` statement returns. For more information about *`bulk_collect_into_clause`*, see "*bulk_collect_into_clause* ::=".

To have the `FETCH` statement retrieve all rows at once, omit `LIMIT` *`numeric_expression`*.

To limit the number of rows that the `FETCH` statement retrieves at once, specify `LIMIT` *`numeric_expression`*.

**Restrictions on *bulk_collect_into_clause***

- You cannot use *`bulk_collect_into_clause`* in client programs.
- When the `FETCH` statement requires implicit data type conversions, *`bulk_collect_into_clause`* can have only one *`collection`* or *`host_array`*.

**Examples**

- Example 6-57, "FETCH Assigns Values to Record that Function Returns"
- Example 7-6, "FETCH Statements Inside LOOP Statements"
- Example 7-7, "Fetching Same Explicit Cursor into Different Variables"
- Example 7-26, "Fetching Data with Cursor Variables"
- Example 7-27, "Fetching from Cursor Variable into Collections"
- Example 7-41, " FETCH with FOR UPDATE Cursor After COMMIT Statement"
- Example 8-8, "Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements"
- Example 13-22, "Bulk-Fetching into Two Nested Tables"
- Example 13-23, "Bulk-Fetching into Nested Table of Records"
- Example 13-24, "Limiting Bulk FETCH with LIMIT"

**Related Topics**

In this chapter:

- "Assignment Statement"
- "CLOSE Statement"
- "Cursor Variable Declaration"
- "Explicit Cursor Declaration and Definition"
- "OPEN Statement"
- "OPEN FOR Statement"
- "RETURNING INTO Clause"
- "%ROWTYPE Attribute"
- "SELECT INTO Statement"

- "%TYPE Attribute"

In other chapters:

- "Using FETCH to Assign a Row to a Record Variable"
- "Fetching Data with Explicit Cursors"
- "Processing Query Result Sets With Cursor FOR LOOP Statements"
- "Fetching Data with Cursor Variables"
- "OPEN FOR, FETCH, and CLOSE Statements"
- "FETCH Statement with BULK COLLECT Clause"
- "Fetching from Results of Pipelined Table Functions"

# FOR LOOP Statement

With each iteration of the `FOR LOOP` statement, its statements run, its index is either incremented or decremented, and control returns to the top of the loop.

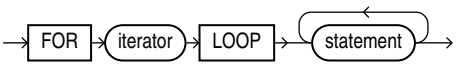
The `FOR LOOP` statement ends when its index reaches a specified value, or when a statement inside the loop transfers control outside the loop or raises an exception. An index is also called an iterand. Statements outside the loop cannot reference the iterand. After the `FOR LOOP` statement runs, the iterand is undefined.

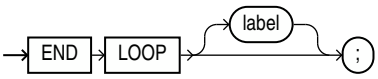
**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***for_loop_statement* ::=**




(Iterator, *statement* ::=)

**Semantics**

***for_loop_statement***

***iterator***

See *iterator*

***statement***

An `EXIT`, `EXIT WHEN`, `CONTINUE`, or `CONTINUE WHEN` in the *`statements`* can cause the loop or the current iteration of the loop to end early. See "*statement* ::=" for the list of all possible statements.

***label***

A label that identifies *`for_loop_statement`* (see "*label*"). `CONTINUE`, `EXIT`, and `GOTO` statements can reference this label.

Labels improve readability, especially when `LOOP` statements are nested, but only if you ensure that the label in the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

**Examples**

- Example 5-20, "Simple Step Filter Using FOR LOOP Stepped Range Iterator"
- Example 5-16, "FOR LOOP Statement Range Iteration Control"
- Example 5-17, "Reverse FOR LOOP Statement Range Iteration Control"
- Example 5-28, "Using FOR LOOP Stopping Predicate Clause"
- Example 5-29, "Using FOR LOOP Skipping Predicate Clause"
- Example 5-11, "Outside Statement References FOR LOOP Statement Index"
- Example 5-12, "FOR LOOP Statement Index with Same Name as Variable"
- Example 5-13, "FOR LOOP Statement References Variable with Same Name as Index"
- Example 5-14, "Nested FOR LOOP Statements with Same Index Name"

**Example 14-28 EXIT WHEN Statement in FOR LOOP Statement**

Suppose that you must exit a `FOR LOOP` statement immediately if a certain condition arises. You can put the condition in an `EXIT WHEN` statement inside the `FOR LOOP` statement.

In this example, the `FOR LOOP` statement is processed 10 times unless the `FETCH` statement inside it fails to return a row, in which case it ends immediately.

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;
  -- Fetch entire row into v_employees record:
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_employees;
    EXIT WHEN c1%NOTFOUND;
    -- Process data here
  END LOOP;
  CLOSE c1;
END;
/
```

**Example 14-29 EXIT WHEN Statement in Inner FOR LOOP Statement**

Now suppose that the `FOR LOOP` statement that you must exit early is nested inside another `FOR LOOP` statement. If, when you exit the inner loop early, you also want to exit the outer loop, then label the outer loop and specify its name in the `EXIT WHEN` statement.

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
```

```
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      EXIT outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

### Example 14-30 CONTINUE WHEN Statement in Inner FOR LOOP Statement

If you want to exit the inner loop early but complete the current iteration of the outer loop, then label the outer loop and specify its name in the `CONTINUE WHEN` statement.

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      CONTINUE outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

### Related Topics

- "FOR LOOP Iterand"
- "FOR LOOP Statement Overview" for more conceptual information
- "Basic LOOP Statement"
- "CONTINUE Statement"
- "Cursor FOR LOOP Statement"
- "EXIT Statement"
- "FETCH Statement"
- "FORALL Statement"
- "OPEN Statement"
- "WHILE LOOP Statement"

- "Overview of Exception Handling" for information about exceptions, which can also cause a loop to end immediately if a certain condition arises

# FORALL Statement

The `FORALL` statement runs one DML statement multiple times, with different values in the `VALUES` and `WHERE` clauses.

The different values come from existing, populated collections or host arrays. The `FORALL` statement is usually much faster than an equivalent `FOR LOOP` statement.

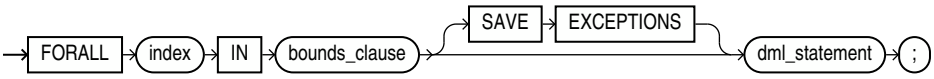
> **Note:**
>
> You can use the `FORALL` statement only in server programs, not in client programs.

**Topics**

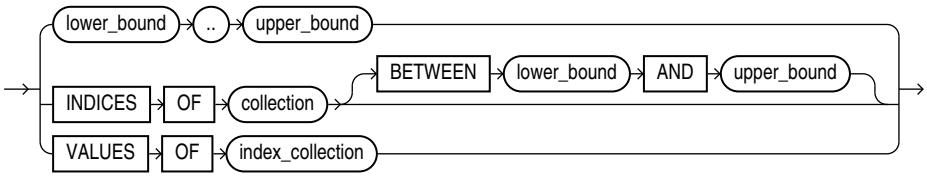
- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***forall_statement* ::=**




***bounds_clause* ::=**




**Semantics**

***forall_statement***

***index***

Name for the implicitly declared integer variable that is local to the `FORALL` statement. Statements outside the `FORALL` statement cannot reference *`index`*. Statements inside the `FORALL` statement can reference *`index`* as an index variable, but cannot use it in expressions or change its value. After the `FORALL` statement runs, *`index`* is undefined.

### *dml_statement*

A static or dynamic `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statement that references at least one collection in its `VALUES` or `WHERE` clause. Performance benefits apply only to collection references that use *`index`* as an index.

Every collection that *`dml_statement`* references must have indexes that match the values of *`index`*. If you apply the `DELETE`, `EXTEND`, or `TRIM` method to one collection, apply it to the other collections also, so that all collections have the same set of indexes. If any collection lacks a referenced element, PL/SQL raises an exception.

#### Restriction on *dml_statement*

If *`dml_statement`* is a dynamic SQL statement, then values in the `USING` clause (bind variables for the dynamic SQL statement) must be simple references to the collection, not expressions. For example, *`collection`*`(`*`i`*`)` is valid, but `UPPER(`*`collection`*`(`*`i`*`))` is invalid.

### SAVE EXCEPTIONS

Lets the `FORALL` statement continue even if some of its DML statements fail. For more information, see "Handling FORALL Exceptions After FORALL Statement Completes".

### *bounds_clause*

Specifies the collection element indexes that provide values for the variable *`index`*. For each value, the SQL engine runs *`dml_statement`* once.

### *lower_bound .. upper_bound*

Both *`lower_bound`* and *`upper_bound`* are numeric expressions that PL/SQL evaluates once, when the `FORALL` statement is entered, and rounds to the nearest integer if necessary. The resulting integers must be the lower and upper bounds of a valid range of consecutive index numbers. If an element in the range is missing or was deleted, PL/SQL raises an exception.

### INDICES OF *collection* [ BETWEEN *lower_bound* AND *upper_bound* ]

Specifies that the values of *`index`* correspond to the indexes of the elements of *`collection`*. The indexes need not be consecutive.

Both *`lower_bound`* and *`upper_bound`* are numeric expressions that PL/SQL evaluates once, when the `FORALL` statement is entered, and rounds to the nearest integer if necessary. The resulting integers are the lower and upper bounds of a valid range of index numbers, which need not be consecutive.

#### Restriction on *collection*

If *`collection`* is an associative array, it must be indexed by `PLS_INTEGER`.

### VALUES OF *index_collection*

Specifies that the values of *`index`* are the elements of *`index_collection`*, a collection of `PLS_INTEGER` elements that is indexed by `PLS_INTEGER`. The indexes of *`index_collection`* need not be consecutive. If *`index_collection`* is empty, PL/SQL raises an exception and the `FORALL` statement does not run.

### Examples

- Example 13-8, "DELETE Statement in FORALL Statement"
- Example 13-9, "Time Difference for INSERT Statement in FOR LOOP and FORALL Statements"

- Example 13-10, "FORALL Statement for Subset of Collection"
- Example 13-11, "FORALL Statements for Sparse Collection and Its Subsets"
- Example 13-12, "Handling FORALL Exceptions Immediately"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"
- Example 13-27, "DELETE with RETURN BULK COLLECT INTO in FORALL Statement"
- Example 13-29, "Anonymous Block Bulk-Binds Input Host Array"

**Related Topics**

In this chapter:

- "FOR LOOP Statement"
- "Implicit Cursor Attribute"

In other chapters:

- "FORALL Statement"
- "BULK COLLECT Clause"
- "Using FORALL Statement and BULK COLLECT Clause Together"

# Formal Parameter Declaration

A formal parameter declaration specifies the name and data type of the parameter, and (optionally) its mode and default value.

A formal parameter declaration can appear in the following:

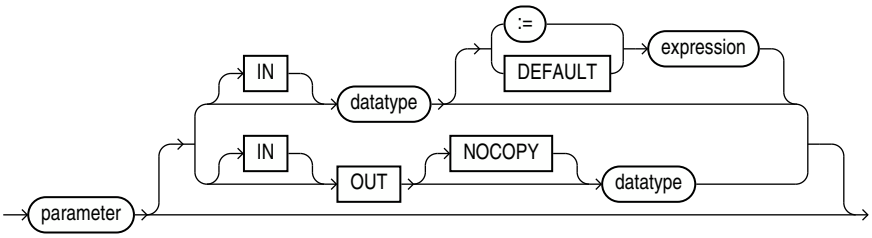
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***parameter_declaration* ::=**



### Semantics

***parameter_declaration***

***parameter***

Name of the formal parameter that you are declaring, which you can reference in the executable part of the subprogram.

**IN, OUT, IN OUT**

Mode that determines the behavior of the parameter, explained in "Subprogram Parameter Modes". **Default:** `IN`.

> **Note:**
>
> Avoid using `OUT` and `IN OUT` for function parameters. The purpose of a function is to take zero or more parameters and return a single value. Functions must be free from side effects, which change the values of variables not local to the subprogram.

**NOCOPY**

Requests that the compiler pass the corresponding actual parameter by reference instead of value (for the difference, see "Subprogram Parameter Passing Methods"). Each time the subprogram is invoked, the optimizer decides, silently, whether to obey or disregard `NOCOPY`.

> **Caution:**
>
> `NOCOPY` increases the likelihood of aliasing. For details, see "Subprogram Parameter Aliasing with Parameters Passed by Reference".

The compiler ignores `NOCOPY` in these cases:

- The actual parameter must be implicitly converted to the data type of the formal parameter.
- The actual parameter is the element of a collection.
- The actual parameter is a scalar variable with the `NOT NULL` constraint.
- The actual parameter is a scalar numeric variable with a range, size, scale, or precision constraint.
- The actual and formal parameters are records, one or both was declared with `%ROWTYPE` or `%TYPE`, and constraints on corresponding fields differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor `FOR LOOP` statement, and constraints on corresponding fields differ.
- The subprogram is invoked through a database link or as an external subprogram.

> **Note:**
>
> The preceding list might change in a subsequent release.

***datatype***

Data type of the formal parameter that you are declaring. The data type can be a constrained subtype, but cannot include a constraint (for example, `NUMBER(2)` or `VARCHAR2(20)`).

If *`datatype`* is a constrained subtype, the corresponding actual parameter inherits the `NOT NULL` constraint of the subtype (if it has one), but not the size (see Example 9-10).

> **Caution:**
>
> The data type `REF CURSOR` increases the likelihood of subprogram parameter aliasing, which can have unintended results. For more information, see "Subprogram Parameter Aliasing with Cursor Variable Parameters".

***expression***

Default value of the formal parameter that you are declaring. The data type of *`expression`* must be compatible with *`datatype`*.

If a subprogram invocation does not specify an actual parameter for the formal parameter, then that invocation evaluates *`expression`* and assigns its value to the formal parameter.

If a subprogram invocation does specify an actual parameter for the formal parameter, then that invocation assigns the value of the actual parameter to the formal parameter and does not evaluate *`expression`*.

**Examples**

- Example 3-26, "Assigning Value to Variable as IN OUT Subprogram Parameter"
- Example 9-9, "Formal Parameters and Actual Parameters"
- Example 9-14, "Parameter Values Before, During, and After Procedure Invocation"
- Example 9-15, "OUT and IN OUT Parameter Values After Exception Handling"
- Example 9-20, "Procedure with Default Parameter Values"
- Example 9-21, "Function Provides Default Parameter Value"
- Example 9-22, "Adding Subprogram Parameter Without Changing Existing Invocations"

**Related Topics**

In this chapter:

- "Function Declaration and Definition"
- "Procedure Declaration and Definition"

In other chapters:

- "Subprogram Parameters"

- "Tune Subprogram Invocations"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"

# Function Declaration and Definition

Before invoking a function, you must declare and define it. You can either declare it first (with ***function_declaration***) and then define it later in the same block, subprogram, or package (with ***function_definition***) or declare and define it at the same time (with ***function_definition***).

A **function** is a subprogram that returns a value. The data type of the value is the data type of the function. A function invocation (or call) is an expression, whose data type is that of the function.

A function declaration is also called a **function specification** or **function spec**.

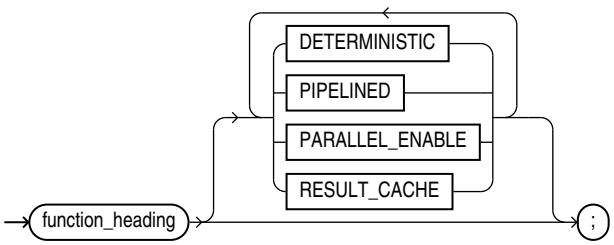
> **Note:**
>
> This topic applies to nested functions.
>
> For information about standalone functions, see "CREATE FUNCTION Statement".
>
> For information about package functions, see "CREATE PACKAGE Statement".

**Topics**

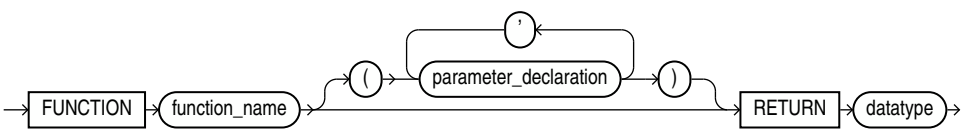
- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***function_declaration* ::=**

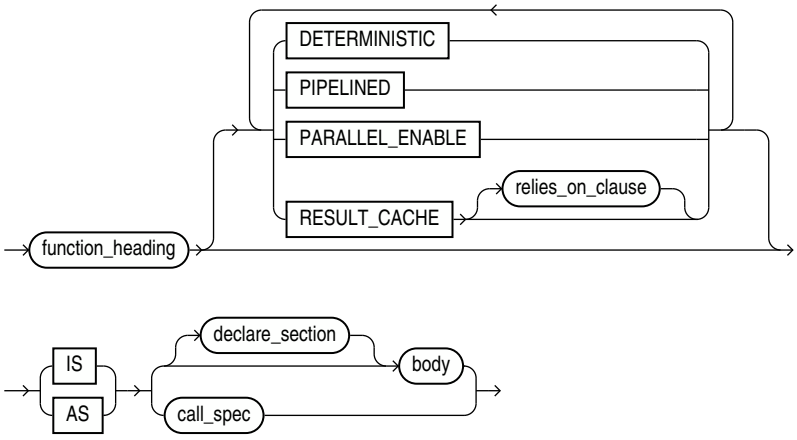



***function_heading* ::=**



( *datatype* ::= , *parameter_declaration* ::= )

***function_definition* ::=**




( *body* ::= , *declare_section* ::= , *pipelined_clause* ::= , *deterministic_clause* ::= , *parallel_enable_clause* ::= , *result_cache_clause* ::= , *call_spec* ::= )

**Semantics**

***function_declaration***

Declares a function, but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.

***function_heading***

The function heading specifies the function name and its parameter list.

***function_name***

Name of the function that you are declaring or defining.

**RETURN *datatype***

Specifies the data type of the value that the function returns, which can be any PL/SQL data type (see PL/SQL Data Types).

**Restriction on *datatype***

You cannot constrain this data type (with `NOT NULL`, for example). If *`datatype`* is a constrained subtype, then the returned value does not inherit the constraints of the subtype (see "Formal Parameters of Constrained Subtypes").

***function_definition***

Either defines a function that was declared earlier or both declares and defines a function.

***declare_section***

Declares items that are local to the function, can be referenced in *`body`*, and cease to exist when the function completes execution.

***body***

Required executable part and optional exception-handling part of the function. In the executable part, at least one execution path must lead to a `RETURN` statement; otherwise, a runtime error occurs.

**Examples**

- Example 9-2, "Declaring, Defining, and Invoking a Simple PL/SQL Function"

**Related Topics**

- Formal Parameter Declaration
- Procedure Declaration and Definition
- PL/SQL Subprograms

# GOTO Statement

The `GOTO` statement transfers control to a labeled block or statement.

If a `GOTO` statement exits a cursor `FOR LOOP` statement prematurely, the cursor closes.

**Restrictions on GOTO Statement**

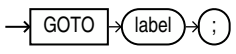
- A `GOTO` statement cannot transfer control into an `IF` statement, `CASE` statement, `LOOP` statement, or sub-block.
- A `GOTO` statement cannot transfer control from one `IF` statement clause to another, or from one `CASE` statement `WHEN` clause to another.
- A `GOTO` statement cannot transfer control out of a subprogram.
- A `GOTO` statement cannot transfer control into an exception handler.
- A `GOTO` statement cannot transfer control from an exception handler back into the current block (but it can transfer control from an exception handler into an enclosing block).

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***goto_statement* ::=**




**Semantics**

***goto_statement***

***label***

Identifies either a block or a statement (see "*plsql_block* ::=", "*statement* ::=", and "*label*").

If *label* is not in the current block, then the GOTO statement transfers control to the first enclosing block in which *label* appears.

### Examples

#### Example 14-31 GOTO Statement

A label can appear before a statement.

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;

  p := ' is a prime number';

  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

Result:

```
37 is a prime number
```

#### Example 14-32 Incorrect Label Placement

A label can only appear before a block or before a statement.

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
       GOTO end_loop;
    END IF;
    <<end_loop>>
  END LOOP;
END;
/
```

Result:

```
  END LOOP;
  *
ERROR at line 9:
ORA-06550: line 9, column 3:
```

```
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( begin case declare exit for goto if loop mod null raise
return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql run commit forall merge pipe purge
```

### Example 14-33 GOTO Statement Goes to Labeled NULL Statement

A label can appear before a `NULL` statement.

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

### Example 14-34 GOTO Statement Transfers Control to Enclosing Block

A `GOTO` statement can transfer control to an enclosing block from the current block.

```
DECLARE
  v_last_name  VARCHAR2(25);
  v_emp_id     NUMBER(6) := 120;
BEGIN
  <<get_name>>
  SELECT last_name INTO v_last_name
  FROM employees
  WHERE employee_id = v_emp_id;

  BEGIN
    DBMS_OUTPUT.PUT_LINE (v_last_name);
    v_emp_id := v_emp_id + 5;

    IF v_emp_id < 120 THEN
      GOTO get_name;
    END IF;
  END;
END;
/
```

Result:

```
Weiss
```

### Example 14-35 GOTO Statement Cannot Transfer Control into IF Statement

The `GOTO` statement transfers control into an `IF` statement, causing an error.

```
DECLARE
  valid BOOLEAN := TRUE;
BEGIN
  GOTO update_row;

  IF valid THEN
  <<update_row>>
    NULL;
  END IF;
END;
/
```

Result:

```
  GOTO update_row;
  *
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00375: illegal GOTO statement; this GOTO cannot branch to label
'UPDATE_ROW'
ORA-06550: line 6, column 12:
PL/SQL: Statement ignored
```

#### Related Topics

- "Block"
- "GOTO Statement"

# IF Statement

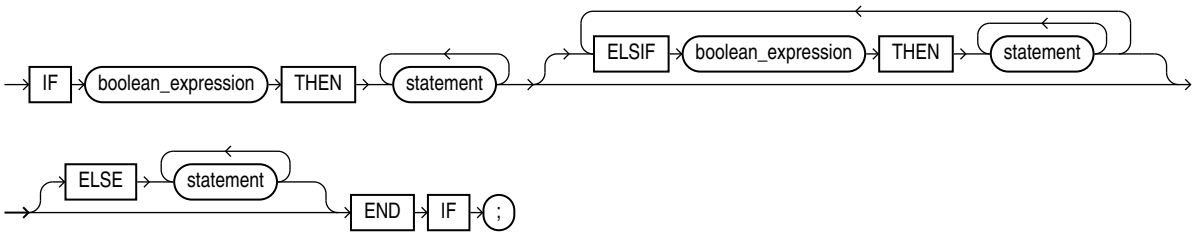
The `IF` statement either runs or skips a sequence of one or more statements, depending on the value of a `BOOLEAN` expression.

#### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***if_statement* ::=**




( *boolean_expression* ::= , *statement* ::= )

### Semantics

***boolean_expression***

Expression whose value is `TRUE`, `FALSE`, or `NULL`.

The first *`boolean_expression`* is always evaluated. Each other *`boolean_expression`* is evaluated only if the values of the preceding expressions are `FALSE`.

If a *`boolean_expression`* is evaluated and its value is `TRUE`, the statements after the corresponding `THEN` run. The succeeding expressions are not evaluated, and the statements associated with them do not run.

**ELSE**

If no *`boolean_expression`* has the value `TRUE`, the statements after `ELSE` run.

### Examples

- Example 5-1, "IF THEN Statement"
- Example 5-2, "IF THEN ELSE Statement"
- Example 5-3, "Nested IF THEN ELSE Statements"
- Example 5-4, "IF THEN ELSIF Statement"

### Related Topics

In this chapter:

- "CASE Statement"
- "Expression"

In other chapters:

- "Conditional Selection Statements"

# Implicit Cursor Attribute

An implicit cursor has attributes that return information about the most recently run `SELECT` or DML statement that is not associated with a named cursor.

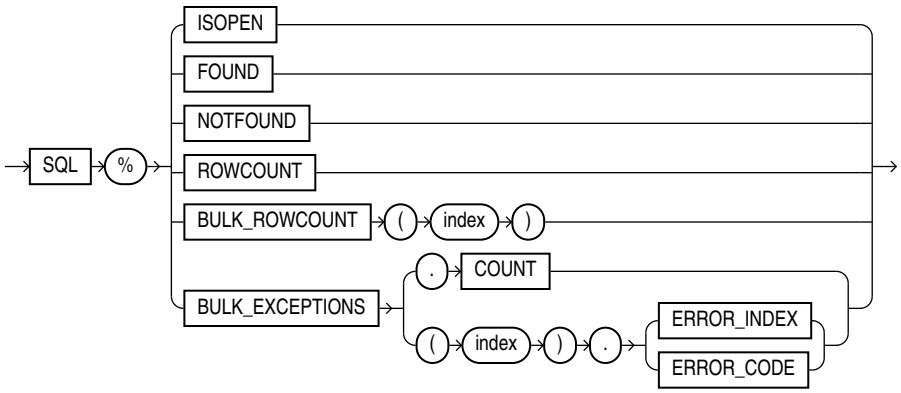
> **Note:**
>
> You can use cursor attributes only in procedural statements, not in SQL statements.

**Topics**


- Syntax
- Semantics
- Examples
- Related Topics


**Syntax**

***implicit_cursor_attribute* ::=**




**Semantics**

**%ISOPEN**

`SQL%ISOPEN` always has the value `FALSE`.

**%FOUND**

`SQL%FOUND` has one of these values:

- If no `SELECT` or DML statement has run, `NULL`.
- If the most recent `SELECT` or DML statement returned a row, `TRUE`.
- If the most recent `SELECT` or DML statement did not return a row, `FALSE`.

### %NOTFOUND

`SQL%NOTFOUND` has one of these values:

- If no `SELECT` or DML statement has run, `NULL`.
- If the most recent `SELECT` or DML statement returned a row, `FALSE`.
- If the most recent `SELECT` or DML statement did not return a row, `TRUE`.

### %ROWCOUNT

`SQL%ROWCOUNT` has one of these values:

- If no `SELECT` or DML statement has run, `NULL`.
- If a `SELECT` or DML statement has run, the number of rows fetched so far.

### SQL%BULK_ROWCOUNT

Composite attribute that is like an associative array whose *i*th element is the number of rows affected by the *i*th DML statement in the most recently completed `FORALL` statement. For more information, see "Getting Number of Rows Affected by FORALL Statement".

### Restriction on SQL%BULK_ROWCOUNT

You cannot assign the value of `SQL%BULK_ROWCOUNT(index)` to another collection.

### SQL%BULK_EXCEPTIONS

Composite attribute that is like an associative array of information about the DML statements that failed during the most recently run `FORALL` statement. `SQL%BULK_EXCEPTIONS.COUNT` is the number of DML statements that failed. If `SQL%BULK_EXCEPTIONS.COUNT` is not zero, then for each index value *i* from 1 through `SQL%BULK_EXCEPTIONS.COUNT`:

- `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX` is the number of the DML statement that failed.
- `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` is the Oracle Database error code for the failure.

Typically, this attribute appears in an exception handler for a `FORALL` statement that has a `SAVE EXCEPTIONS` clause. For more information, see "Handling FORALL Exceptions After FORALL Statement Completes".

### Examples

- Example 7-3, "SQL%FOUND Implicit Cursor Attribute"
- Example 7-4, "SQL%ROWCOUNT Implicit Cursor Attribute"
- Example 7-15, "%FOUND Explicit Cursor Attribute"
- Example 7-14, "%ISOPEN Explicit Cursor Attribute"
- Example 7-16, "%NOTFOUND Explicit Cursor Attribute"
- Example 7-17, "%ROWCOUNT Explicit Cursor Attribute"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"
- Example 13-14, "Showing Number of Rows Affected by Each DELETE in FORALL"
- Example 13-15, "Showing Number of Rows Affected by Each INSERT SELECT in FORALL"

**Related Topics**

In this chapter:

- "FORALL Statement"
- "Named Cursor Attribute"

In other chapters:

- "Implicit Cursors"
- "Processing Query Result Sets"

# INLINE Pragma

The `INLINE` pragma specifies whether a subprogram invocation is to be inlined.

Inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit).

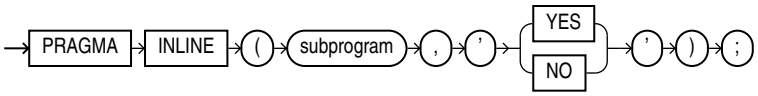
> **Note:**
>
> The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements. For details, see "Subprogram Inlining".

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***inline_pragma* ::=**




**Semantics**

***subprogram***

Name of a subprogram. If *`subprogram`* is overloaded, then the `INLINE` pragma applies to every subprogram with that name.

**YES**

If `PLSQL_OPTIMIZE_LEVEL=2`, `'YES'` specifies that the subprogram invocation is to be inlined.

If `PLSQL_OPTIMIZE_LEVEL=3`, `'YES'` specifies that the subprogram invocation has a high priority for inlining.

**NO**

Specifies that the subprogram invocation is not to be inlined.

**Examples**

- Example 13-1, "Specifying that Subprogram Is To Be Inlined"
- Example 13-2, "Specifying that Overloaded Subprogram Is To Be Inlined"
- Example 13-3, "Specifying that Subprogram Is Not To Be Inlined"
- Example 13-4, "PRAGMA INLINE ... 'NO' Overrides PRAGMA INLINE ... 'YES'"

**Related Topics**

- "Subprogram Inlining"

# Invoker's Rights and Definer's Rights Clause

Specifies the `AUTHID` property of a stored PL/SQL subprogram. The `AUTHID` property affects the name resolution and privilege checking of SQL statements that the unit issues at run time.

The *invoker_rights_clause* can appear in the following SQL statements :

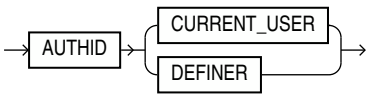
- ALTER TYPE Statement
- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PROCEDURE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

**Topics**

- Syntax
- Semantics
- Related Topics

**Syntax**

***invoker_rights_clause* ::=**




**Semantics**

***invoker_rights_clause***

When it appears in the package declaration, it specifies the `AUTHID` property of functions and procedures in the package, and of the explicit cursors declared in the package specification.

When it appears in a standalone function declaration, it specifies the `AUTHID` property of the function.

When it appears in a standalone procedure declaration, it specifies the `AUTHID` property of the procedure.

The *invoker_rights_clause* can appear only once in a subprogram declaration.

When it appears in an ADT, it specifies the `AUTHID` property of the member functions and procedures of the ADT.

**Restrictions on *invoker_rights_clause***

The following restrictions apply for types:

- This clause is valid only for ADTs, not for a nested table or `VARRAY` type.
- You can specify this clause for clarity if you are creating a subtype. However, a subtype inherits the `AUTHID` property of its supertype, so you cannot specify a different value than was specified for the supertype.
- If the supertype was created with `AUTHID DEFINER`, then you must create the subtype in the same schema as the supertype.
- You cannot specify the `AUTHID` property of SQL macros. They behave like IR units.

**Related Topics**

In this book:

- "Invoker's Rights and Definer's Rights (AUTHID Property)"for information about the `AUTHID` property
- "Subprogram Properties"

# INSERT Statement Extension

The PL/SQL extension to the SQL `INSERT` statement lets you specify a record name in the *`values_clause`* of the *`single_table_insert`* instead of specifying a column list in the *`insert_into_clause`*

Effectively, this form of the `INSERT` statement inserts the record into the table; actually, it adds a row to the table and gives each column of the row the value of the corresponding record field.

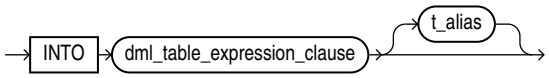
> **See Also:**
>
> *Oracle Database SQL Language Reference* for the syntax of the SQL `INSERT` statement

**Topics**

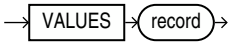
- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***insert_into_clause* ::=**




***values_clause* ::=**




### Semantics

***insert_into_clause***

***dml_table_expression_clause***

Typically a table name. For complete information, see *Oracle Database SQL Language Reference*.

***t_alias***

An alias for *dml_table_expression_clause*.

***values_clause***

***record***

Name of a record variable of type `RECORD` or `%ROWTYPE`. *record* must represent a row of the item explained by *dml_table_expression_clause*. That is, for every column of the row, the record must have a field with a compatible data type. If a column has a `NOT NULL` constraint, then its corresponding field cannot have a `NULL` value.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for the complete syntax of the `INSERT` statement

### Examples

- Example 6-60, "Initializing Table by Inserting Record of Default Values"

### Related Topics

In this chapter:

- "Record Variable Declaration"
- "%ROWTYPE Attribute"

In other chapters:

- "Inserting Records into Tables"
- "Restrictions on Record Inserts and Updates"

# Iterator

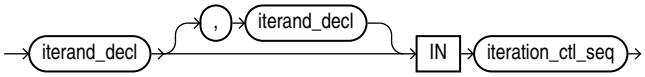
The iterator specifies an iterand and the iteration controls.

An iterator can appear in the following statements:

- FOR LOOP Statement
- Qualified Expression

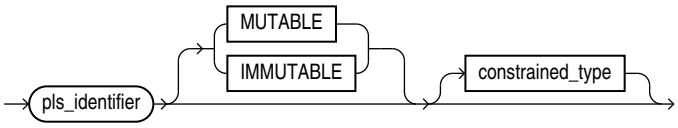
**Syntax**

***iterator* ::=**

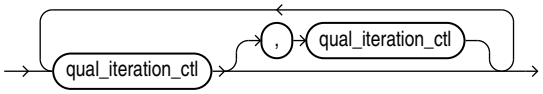



(*iterand_decl* ::=, *iteration_ctl_seq* ::= )

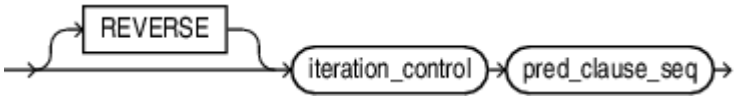
***iterand_decl* ::=**




***iteration_ctl_seq* ::=**

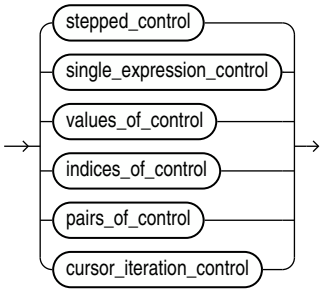



***qual_iteration_ctl* ::=**




(*iteration_control* ::=, *pred_clause_seq* ::=)

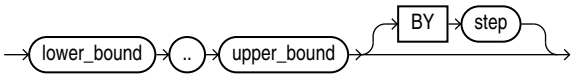
***iteration_control* ::=**



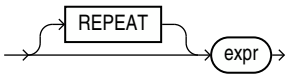

( *stepped_control* ::=, *single_expression_control* ::= , *values_of_control* ::=, *indices_of_control* ::=, *pairs_of_control* ::=, *cursor_iteration_control* ::=)

***pred_clause_seq* ::=**

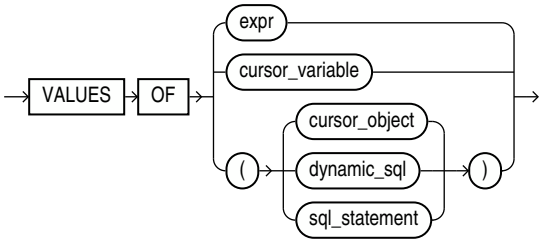

***stepped_control* ::=**




***single_expression_control* ::=**

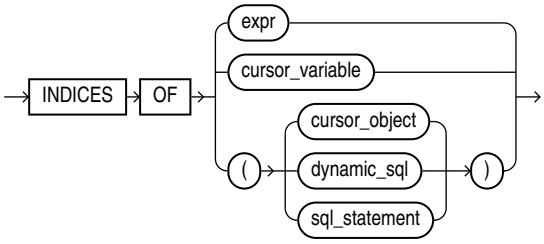



***values_of_control* ::=**



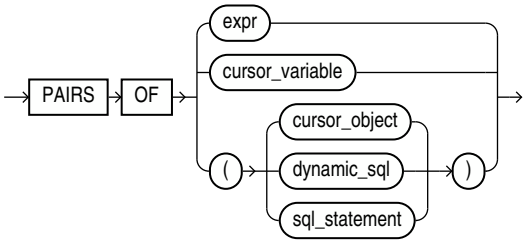

(*dynamic_sql* ::=)

***indices_of_control* ::=**



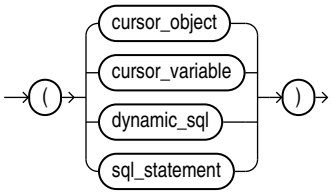

(*dynamic_sql* ::=)

***pairs_of_control* ::=**



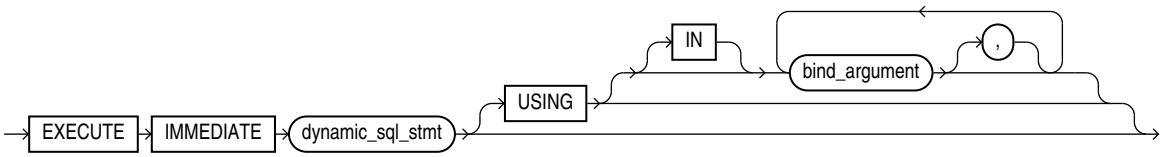

(*dynamic_sql* ::=)

***cursor_iteration_control* ::=**




(*dynamic_sql* ::=)

***dynamic_sql* ::=**



### Semantics

#### *iterator*

The iterator specifies an iterand and the iteration controls.

Statements outside the loop cannot reference *iterator*. Statements inside the loop can reference *iterator*, but cannot change its value. After the FOR LOOP statement runs, *iterator* is undefined.

#### *iterand_decl*

An iterand type can be implicitly or explicitly declared. You cannot explicitly initialize an iterand.

An iterand type is implicitly declared when no type declaration follows the iterand in the loop header. The implicit type is determined by the first iteration control.

**Table 14-1 Iterand Implicit Type Defaults**

| Iteration Control | Implicit Iterand Type |
|---|---|
| stepped control | PLS_INTEGER |
| single expression | PLS_INTEGER |
| cursor control | CURSOR%ROWTYPE |
| VALUES OF control | collection element type |
| INDICES OF control | collection index type |
| PAIRS OF control | The first iterand denotes the index type of collection and the second iterand denotes the element type of collection |

#### *pls_identifier*

The iterand name for the implicitly declared variable that is local to the FOR LOOP statement.

#### [ MUTABLE | IMMUTABLE ]

The mutability property of an iterand determines whether or not it can be assigned in the loop body. If all iteration controls specified in an iterator are cursor controls, the iterand is mutable by default. Otherwise, the iterand is immutable. The default mutability property of an iterand can be changed in the iterand declaration by specifying the MUTABLE or IMMUTABLE keyword after the iterand variable. The mutability property keywords are not reserved and could be used as type names. Such usage would be ambiguous. Therefore, you must explicitly specify the mutability property of an iterand in the iterand declaration if its type is named mutable or immutable. Iterand for INDICES OF iteration control and the index iterand for PAIRS OF iteration control cannot be made mutable.

#### *constrained_type*

An iterand is explicitly declared when the iterand type is specified in the loop header. Any constraint defined for a type is considered when assigning values to the iterand. The values generated by the iteration controls must be assignment compatible with the iterand type. Usual conversion rules apply. Exceptions are raised for all constraint violations.

#### *iteration_ctl_seq*

Multiple iteration controls may be chained together by separating them with commas.

**Restriction on *iteration_ctl_seq*:**

Because two iterands are required for the pairs of iterand, pairs of iteration controls may not be mixed with other kinds of iteration controls.

***qual_iteration_ctl***

The qualified iteration control specifies the REVERSE option and the optional stopping and skipping predicates clauses.
**[ REVERSE ]**

When the optional keyword `REVERSE` is specified, the order of values in the sequence is reversed.

You can use this option with a collection vector value expression. In that case, specifying `REVERSE` generates values from `LAST` to `FIRST` rather than from `FIRST` to `LAST`.

Restrictions on `REVERSE`:

- You cannot use this option when a pipelined function is specified in the iteration control.
- You cannot use this option with single expression iteration control since it generates a single value and therefore the keyword does not have any sensible meaning for this control.
- You cannot use this option when the iteration control specifies a SQL statement. This creates a sequence of records returned by the query. You can specify an `ORDER BY` clause on the SQL statement to sort the rows in the appropriate order.
- You cannot use this option when the collection is a cursor,cursor variable, dynamic SQL, or is an expression that calls a pipelined table function.

***iteration_control***

An iteration control provides a sequence of values to the iterand.

***pred_clause_seq***

An iteration control may be modified with an optional stopping predicate clause followed by an optional skipping predicate clause. The expressions in the predicates must have a `BOOLEAN` type.

**[ WHILE *boolean_expression* ]**

A stopping predicate clause can cause the iteration control to be exhausted. The *boolean_expression* is evaluated at the beginning of each iteration of the loop. If it fails to evaluate to `TRUE`, the iteration control is exhausted.

**[ WHEN *boolean_expression* ]**

A skipping predicate clause can cause the loop body to be skipped for some values. The *boolean_expression* is evaluated. If it fails to evaluate to `TRUE`, the iteration control skips to the next value.

***stepped_control***

***lower_bound .. upper_bound* [ BY *step* ]**

Without `REVERSE`, the value of *`iterand`* starts at *`lower_bound`* and increases by *`step`* with each iteration of the loop until it reaches *`upper_bound`*.

With REVERSE, the value of *iterand* starts at *upper_bound* and decreases by *step* with each iteration of the loop until it reaches *lower_bound*. If *upper_bound* is less than *lower_bound*, then the *statements* never run.

The default value for *step* is one if this optional BY clause is not specified.

*lower_bound* and *upper_bound* must evaluate to numbers (either numeric literals, numeric variables, or numeric expressions). If a bound does not have a numeric value, then PL/SQL raises the predefined exception VALUE_ERROR. PL/SQL evaluates *lower_bound* and *upper_bound* once, when the FOR LOOP statement is entered, and stores them as temporary PLS_INTEGER values, rounding them to the nearest integer if necessary.

If *lower_bound* equals *upper_bound*, the *statements* run only once.

The *step* value must be greater than zero.

***single_expression_control***

A single expression iteration control generates a single value. If REPEAT is specified, the expression will be evaluated repeatedly generating a sequence of values until a stopping clause causes the iteration control to be exhausted.

**Restrictions on *single_expression_control*:**

REVERSE is not allowed for a single expression iteration control.

***values_of_control***

The element type of a collection must be assignment compatible with the iterand.

***indices_of_control***

The index type of a collection must be assignment compatible with the iterand.

The iterand used with an INDICES OF iteration control cannot be mutable.

***pairs_of_control***

The PAIRS OF iteration control requires two iterands. You cannot mix the PAIRS OF iteration control with other kinds of controls. The first iterand is the index iterand and the second is the value iterand. Each iterand may be followed by an explicit type.

The element type of the collection must be assignment compatible with the value iterand. The index type of the collection must be assignment compatible with the index iterand.

The index iterand used with a PAIRS OF iteration control cannot be mutable.

***cursor_iteration_control***

Cursor iteration controls generate the sequence of records returned by an explicit or implicit cursor. The cursor definition is the controlling expression.

**Restrictions on *cursor_iteration_control*:**

You cannot use REVERSE with a cursor iteration control.

***cursor_object***

A *cursor_object* is an explicit PL/SQL cursor object.

***sql_statement***

A *sql_statement* is an implicit PL/SQL cursor object created for a SQL statement specified directly in the iteration control.

***cursor_variable***

Name of a previously declared variable of a `REF CURSOR` object.

***dynamic_sql***

**EXECUTE IMMEDIATE *dynamic_sql_stmt* [ USING [ IN ] (*bind_argument* [,] )+]**

You can use a dynamic query in place of an implicit cursor definition in a cursor or collection iteration control. Such a construct cannot provide a default type; if it is used as the first iteration control, an explicit type must be specified for the iterand, or for the value iterand for a pairs of control.

The optional `USING` clause is the only clause allowed with the dynamic SQL. It can only possibly have IN one or more bind variable, each separated by a comma.

***dynamic_sql_stmt***

String literal, string variable, or string expression that represents a SQL statement. Its type must be either `CHAR`, `VARCHAR2`, or `CLOB`.

> **Caution:**
>
> When using dynamic SQL, beware of SQL injection, a security risk. For more information about SQL injection, see "SQL Injection".

**Examples**

- Example 5-26, "Using Dynamic SQL as an Iteration Control"
- Example 5-18, "Stepped Range Iteration Controls"
- Example 5-19, "STEP Clause in FOR LOOP Statement"
- Example 5-25, "Cursor Iteration Controls"
- Example 5-22, "VALUES OF Iteration Control"
- Example 5-23, "INDICES OF Iteration Control"
- Example 5-24, "PAIRS OF Iteration Control"

**Related Topics**

- FOR LOOP Statement Overview
- Qualified Expressions Overview

# Named Cursor Attribute

Every named cursor (explicit cursor or cursor variable) has four attributes, each of which returns information about the execution of a DML statement.

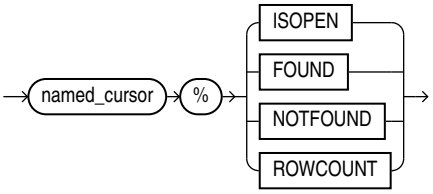
> **Note:**
>
> You can use cursor attributes only in procedural statements, not in SQL statements.

**Topics**

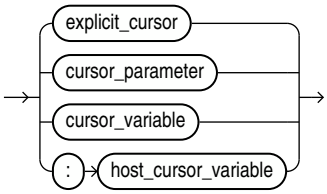

- Syntax
- Semantics
- Examples
- Related Topics


**Syntax**

***named_cursor_attribute* ::=**




***named_cursor* ::=**




**Semantics**

***named_cursor_attribute***

**%ISOPEN**

*named_cursor*`%ISOPEN` has the value `TRUE` if the cursor is open, and `FALSE` if it is not open.

**%FOUND**

*named_cursor*`%FOUND` has one of these values:

- If the cursor is not open, `INVALID_CURSOR`
- If cursor is open but no fetch was tried, `NULL`.
- If the most recent fetch returned a row, `TRUE`.
- If the most recent fetch did not return a row, `FALSE`.

**%NOTFOUND**

*named_cursor*`%NOTFOUND` has one of these values:

- If cursor is not open, `INVALID_CURSOR`.
- If cursor is open but no fetch was tried, `NULL`.
- If the most recent fetch returned a row, `FALSE`.
- If the most recent fetch did not return a row, `TRUE`.

**%ROWCOUNT**

*named_cursor*`%ROWCOUNT` has one of these values:

- If cursor is not open, `INVALID_CURSOR`.
- If cursor is open, the number of rows fetched so far.

***named_cursor***

***explicit_cursor***

Name of an explicit cursor.

***cursor_parameter***

Name of a formal cursor parameter.

***cursor_variable***

Name of a cursor variable.

**:*host_cursor_variable***

Name of a cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *host_cursor_variable*.

### Examples

- Example 7-14, "%ISOPEN Explicit Cursor Attribute"
- Example 7-15, "%FOUND Explicit Cursor Attribute"
- Example 7-16, "%NOTFOUND Explicit Cursor Attribute"
- Example 7-17, "%ROWCOUNT Explicit Cursor Attribute"

### Related Topics

In this chapter:

- "Cursor Variable Declaration"
- "Explicit Cursor Declaration and Definition"
- "Implicit Cursor Attribute"

In other chapters:

- "Explicit Cursor Attributes"

# NULL Statement

The `NULL` statement is a "no-op" (no operation)—it only passes control to the next statement.

> **Note:**
>
> The `NULL` statement and the `BOOLEAN` value `NULL` are not related.

**Topics**

- Syntax
- Examples
- Related Topics

**Syntax**

***null_statement* ::=**


**Examples**

- Example 5-30, "NULL Statement Showing No Action"
- Example 5-31, "NULL Statement as Placeholder During Subprogram Creation"

**Related Topics**

- "NULL Statement"

# OPEN Statement

The `OPEN` statement opens an explicit cursor, allocates database resources to process the associated query, identifies the result set, and positions the cursor before the first row of the result set.

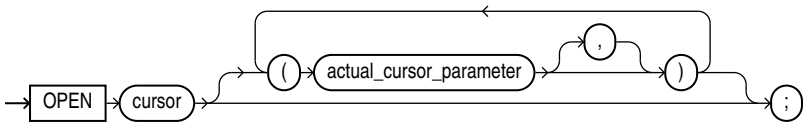
If the query has a `FOR UPDATE` clause, the `OPEN` statement locks the rows of the result set.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***open_statement* ::=**




### Semantics

***cursor***

Name of an explicit cursor that is not open.

***actual_cursor_parameter***

List of actual parameters for the cursor that you are opening. An actual parameter can be a constant, initialized variable, literal, or expression. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter.

You can specify actual cursor parameters with either positional notation or named notation. For information about these notations, see "Positional, Named, and Mixed Notation for Actual Parameters".

If the cursor specifies a default value for a parameter, you can omit that parameter from the parameter list. If the cursor has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

### Examples

- Example 7-11, "Explicit Cursor that Accepts Parameters"
- Example 7-12, "Cursor Parameters with Default Values"

### Related Topics

In this chapter:

- "CLOSE Statement"
- "Explicit Cursor Declaration and Definition"
- "FETCH Statement"
- "OPEN FOR Statement"

In other chapters:

- "Opening and Closing Explicit Cursors"
- "Explicit Cursors that Accept Parameters"

# OPEN FOR Statement

The `OPEN FOR` statement associates a cursor variable with a query, allocates database resources to process the query, identifies the result set, and positions the cursor before the first row of the result set.

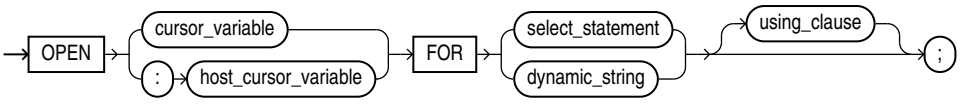
If the query has a `FOR UPDATE` clause, then the `OPEN FOR` statement locks the rows of the result set.

**Topics**

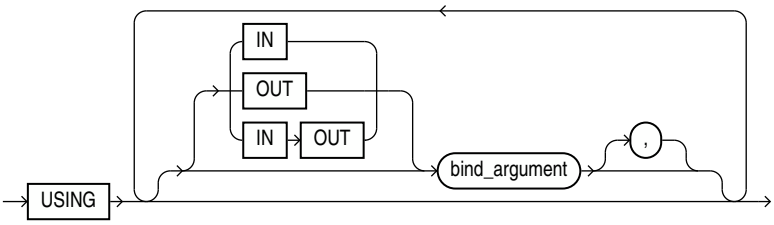

- Syntax
- Semantics
- Examples
- Related Topics


**Syntax**

***open_for_statement* ::=**




***using_clause* ::=**




**Semantics**

***open_for_statement***

***cursor_variable***

Name of a cursor variable. If *`cursor_variable`* is the formal parameter of a subprogram, then it must not have a return type. For information about cursor variables as subprogram parameters, see "Cursor Variables as Subprogram Parameters".

***:host_cursor_variable***

Name of a cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *`host_cursor_variable`*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

***select_statement***

SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement). Typically, *`select_statement`* returns multiple rows.

> **See:**
>
> *Oracle Database SQL Language Reference* for `SELECT` statement syntax

***dynamic_string***

String literal, string variable, or string expression of the data type `CHAR`, `VARCHAR2`, or `CLOB`, which represents a SQL `SELECT` statement. Typically, *`dynamic_statement`* represents a SQL `SELECT` statement that returns multiple rows.

***using_clause***

Specifies bind variables, using positional notation.

> **Note:**
>
> If you repeat placeholder names in *`dynamic_sql_statement`*, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement. For details, see "Repeated Placeholder Names in Dynamic SQL Statements."

**Restriction on *using_clause***

Use if and only if *`select_statement`* or *`dynamic_sql_stmt`* includes placeholders for bind variables.

**IN, OUT, IN OUT**

Parameter modes of bind variables. An `IN` bind variable passes its value to the *`select_statement`* or *`dynamic_string`*. An `OUT` bind variable stores a value that *`dynamic_string`* returns. An `IN OUT` bind variable passes its initial value to *`dynamic_string`* and stores a value that *`dynamic_string`* returns. **Default:** `IN`.

***bind_argument***

Expression whose value replaces its corresponding placeholder in *`select_statement`* or *`dynamic_string`* at run time. You must specify a *`bind_argument`* for every placeholder.

> **Note:**
>
> Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.

### Restrictions on *bind_argument*

- *bind_argument* cannot be an associative array indexed by string.
- *bind_argument* cannot be the reserved word NULL.

  To pass the value NULL to the dynamic SQL statement, use an uninitialized variable where you want to use NULL, as in Example 8-7.

### Examples

- Example 7-26, "Fetching Data with Cursor Variables"
- Example 7-30, "Querying a Collection with Static SQL"
- Example 7-31, "Procedure to Open Cursor Variable for One Query"
- Example 7-32, "Opening Cursor Variable for Chosen Query (Same Return Type)"
- Example 7-33, "Opening Cursor Variable for Chosen Query (Different Return Types)"
- Example 8-8, "Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements"
- Example 8-9, "Querying a Collection with Native Dynamic SQL"

### Related Topics

In this chapter:

- "CLOSE Statement"
- "Cursor Variable Declaration"
- "EXECUTE IMMEDIATE Statement"
- "FETCH Statement"
- "OPEN Statement"

In other chapters:

- "Opening and Closing Cursor Variables"
- "OPEN FOR, FETCH, and CLOSE Statements"

# PARALLEL_ENABLE Clause

Enables the function for parallel execution, making it safe for use in concurrent sessions of parallel DML evaluations.

Indicates that the function can run from a parallel execution server of a parallel query operation.

The PARALLEL_ENABLE clause can appear in the following SQL statements:

- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE TYPE BODY Statement

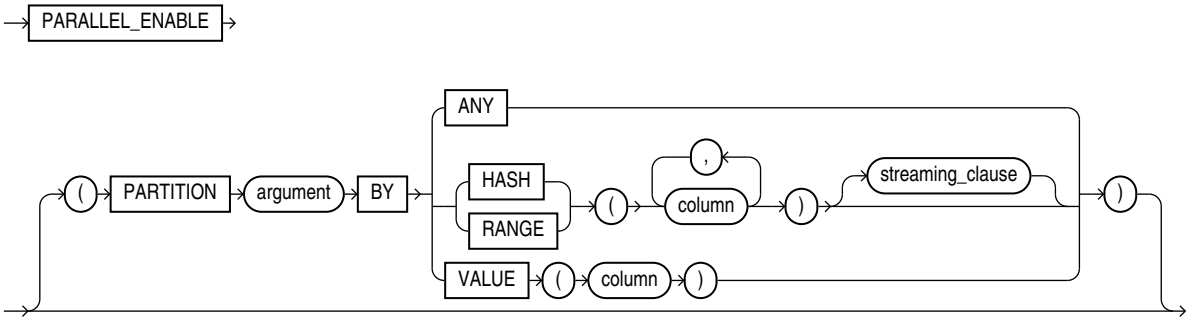
### Topics

- Syntax
- Semantics

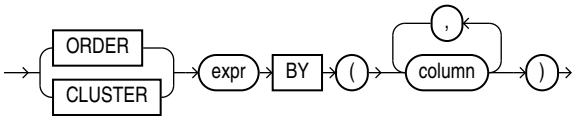
- Related Topics

**Syntax**

***parallel_enable_clause* ::=**




***streaming_clause* ::=**




**Semantics**

***parallel_enable_clause***

The *parallel_enable_clause* can appear only once in the function.

The function must not use session state, such as package variables, because those variables are not necessarily shared among the parallel execution servers.

Use the optional `PARTITION` *`argument`* `BY` clause only with a function that has a `REF CURSOR` data type. This clause lets you define the partitioning of the inputs to the function from the `REF CURSOR` argument. Partitioning the inputs to the function affects the way the query is parallelized when the function is used as a table function in the `FROM` clause of the query.

**ANY**

Indicates that the data can be partitioned randomly among the parallel execution servers

> **Note:**
>
> You can partition weak cursor variable arguments to table functions only with `ANY`, not with `RANGE`, `HASH`, or `VALUE`.

**RANGE or HASH**

Partitions data into specified columns that are returned by the `REF CURSOR` argument of the function.

***streaming_clause***

The optional *streaming_clause* lets you order or cluster the parallel processing.

**ORDER BY | CLUSTER BY**

ORDER BY or CLUSTER BY indicates that the rows on a parallel execution server must be locally ordered and have the same key values as specified by the *column* list.

**VALUE**

Specifies direct-key partitioning, which is intended for table functions used when executing MapReduce workloads. The *column* must be of data type NUMBER. VALUE distributes row processing uniformly over the available reducers.

If the column has more reducer numbers than there are available reducers, then PL/SQL uses a modulus operation to map the reducer numbers in the column into the correct range.

When calculating the number of the reducer to process the corresponding row, PL/SQL treats a negative value as zero and rounds a positive fractional value to the nearest integer.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information about using parallel table functions

***expr***

*expr* identifies the REF CURSOR parameter name of the table function on which partitioning was specified, and on whose columns you are specifying ordering or clustering for each concurrent session in a parallel query execution.

**Restriction on *parallel_enable_clause***

You cannot specify the *parallel_enable_clause* for a nested function or SQL macro.

**Related Topics**

In this chapter:

- Function Declaration and Definition

In other chapters:

- Overview of Table Functions
- Creating Pipelined Table Functions

# PIPE ROW Statement

The `PIPE ROW` statement, which can appear only in the body of a pipelined table function, returns a table row (but not control) to the invoker of the function.

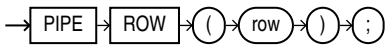
> **Note:**
>
> - If a pipelined table function is part of an autonomous transaction, then it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement, to avoid an error in the invoking subprogram.
> - To improve performance, the PL/SQL runtime system delivers the piped rows to the invoker in batches.

**Topics**


- Syntax
- Semantics
- Examples
- Related Topics


**Syntax**

***pipe_row_statement* ::=**




**Semantics**

***pipe_row_statement***

***row***

Row (table element) that the function returns to its invoker, represented by an expression whose type is that of the table element.

If the expression is a record variable, it must be explicitly declared with the data type of the table element. It cannot be declared with a data type that is only structurally identical to the element type. For example, if the element type has a name, then the record variable cannot be declared explicitly with `%TYPE` or `%ROWTYPE` or implicitly with `%ROWTYPE` in a cursor `FOR LOOP` statement.

**Examples**

- Example 13-30, "Creating and Invoking Pipelined Table Function"
- Example 13-31, "Pipelined Table Function Transforms Each Row to Two Rows"
- Example 13-33, "Pipelined Table Function with Two Cursor Variable Parameters"
- Example 13-34, "Pipelined Table Function as Aggregate Function"

- Example 13-35, "Pipelined Table Function Does Not Handle NO_DATA_NEEDED"
- Example 13-36, "Pipelined Table Function Handles NO_DATA_NEEDED"

**Related Topics**

In this chapter:

- "Function Declaration and Definition"

In other chapters:

- "Creating Pipelined Table Functions"

# PIPELINED Clause

Instructs the database to iteratively return the results of a **table function** or **polymorphic table function** .

Use only with a table function, to specify that it is pipelined. A pipelined table function returns a row to its invoker immediately after processing that row and continues to process rows. To return a row (but not control) to the invoker, the function uses the "PIPE ROW Statement".

A table function returns a collection type.

A polymorphic table function is a table function whose return type is determined by the arguments.

You query both kinds of table functions by using the `TABLE` keyword before the function name in the `FROM` clause of the query. For example:

```
SELECT * FROM TABLE(function_name(...))
```

The `TABLE` operator is optional when the table function arguments list or empty list () appears. For example:

```
SELECT * FROM function_name()
```

the database then returns rows as they are produced by the function.

The PIPELINED option can appear in the following SQL statements:

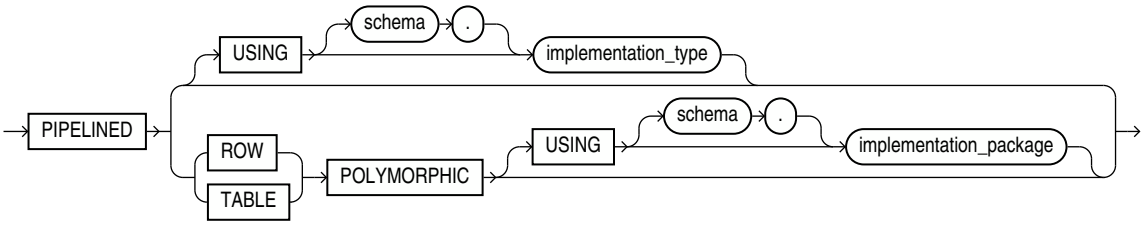
- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***pipelined_clause* ::=**




### Semantics

***pipelined_clause***

The *pipelined_clause* can appear only once in the function.

**PIPELINED**

To make a pipelined function, include the *pipelined_clause* in the function definition. If you declare the pipelined function before defining it, you must specify the `PIPELINED` option in the function declaration.

**{ IS | USING }**

- If you specify the keyword `PIPELINED` alone (`PIPELINED IS` ...), then the PL/SQL function body must use the `PIPE` keyword. This keyword instructs the database to return single elements of the collection out of the function, instead of returning the whole collection as a single value.
- You can specify the `PIPELINED USING` *`implementation_type`* clause to predefine an interface containing the start, fetch, and close operations. The implementation type must implement the `ODCITable` interface and must exist at the time the table function is created. This clause is useful for table functions implemented in external languages such as C++ and Java.

  If the return type of the function is `ANYDATASET`, then you must also define a describe method (`ODCITableDescribe`) as part of the implementation type of the function.

***[schema.] implementation_type***

The implementation type must be an ADT containing the implementation of the `ODCIAggregate` subprograms. If you do not specify *`schema`*, then the database assumes that the implementation type is in your schema.

**Restriction on PIPELINED**

You cannot specify `PIPELINED` for a nested function or a SQL macro.

> **Note:**
>
> You cannot run a pipelined table function over a database link. The reason is that the return type of a pipelined table function is a SQL user-defined type, which can be used only in a single database (as explained in *Oracle Database Object-Relational Developer's Guide*). Although the return type of a pipelined table function might appear to be a PL/SQL type, the database actually converts that PL/SQL type to a corresponding SQL user-defined type.

***PIPELINED [ ROW | TABLE ] POLYMORPHIC [ USING [schema.] implementation_package ]***

The polymorphic table function elaborator can appear in standalone function declaration or package function declaration.

**PIPELINED**

Required when defining a polymorphic table function.

**ROW**

Specify `ROW` when a single input argument of type `TABLE` determines new columns using any single row.

**TABLE**

Specify `TABLE` when a single input argument of type `TABLE` determines the new columns using the current row and operates on an entire table or a logical partition of a table.

**POLYMORPHIC**

**Restrictions on `POLYMORPHIC`**

The following are not allowed for `POLYMORPHIC` table functions:

- PARALLEL_ENABLE clause
- RESULT_CACHE clause
- DETERMINISTIC option
- AUTHID property (Invoker's Rights and Definer's Rights Clause)

**[USING *[schema.] implementation_package*]**

References the polymorphic table function (PTF) implementation package. The specification must include `DESCRIBE` method. The specification of `OPEN`, `FETCH_ROWS` and `CLOSE` methods is optional. The specification for the implementation package must already exist (unless the PTF and its implementation reside in the same package).

If a polymorphic table function and its implementation methods are defined in the same package, then the `USING` clause is optional.

**Examples**

- Examples for PIPE ROW statement examples
- Skip_col Polymorphic Table Function Example
- To_doc Polymorphic Table Function Example

- Implicit_echo Polymorphic Table Function Example
- *Oracle Database PL/SQL Packages and Types Reference* for more examples using the DBMS_TF package utilities

**Related Topics**

In this chapter:

- "Function Declaration and Definition"

In other chapters:

- "Overview of Table Functions"
- "Overview of Polymorphic Table Functions" for more information about PTFs
- "Subprogram Parts"
- "Creating Pipelined Table Functions"
- "Chaining Pipelined Table Functions for Multiple Transformations"

In other books:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_TF package containing utilities for Polymorphic Table Functions (PTF) implementation
- *Oracle Database Data Cartridge Developer's Guide* for information about using pipelined table functions

# Procedure Declaration and Definition

Before invoking a procedure, you must declare and define it. You can either declare it first (with ***procedure_declaration***) and then define it later in the same block, subprogram, or package (with ***procedure_definition***) or declare and define it at the same time (with ***procedure_definition***).

A **procedure** is a subprogram that performs a specific action. A procedure invocation (or call) is a statement.

A procedure declaration is also called a **procedure specification** or **procedure spec**.

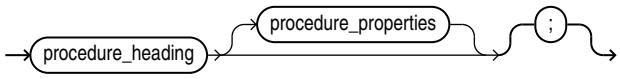
> **Note:**
>
> For more information about standalone procedures, see "CREATE PROCEDURE Statement". For more information about package procedures, see "CREATE PACKAGE Statement".

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

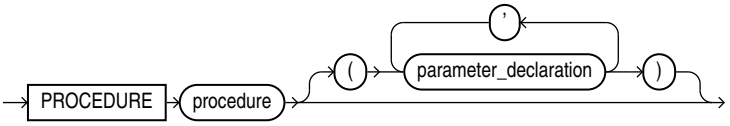
### Syntax

***procedure_declaration* ::=**



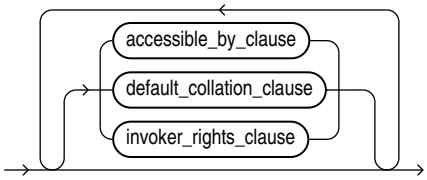

( *procedure_properties* ::= )

***procedure_heading* ::=**



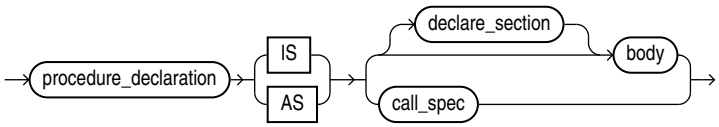

See "*parameter_declaration* ::=".

***procedure_properties* ::=**




( *accessible_by_clause* ::= , *default_collation_clause* ::= , *invoker_rights_clause* ::= )

***procedure_definition* ::=**




( *body* ::= , *declare_section* ::= , *call_spec* ::= )

### Semantics

***procedure_declaration***

Declares a procedure, but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.

***procedure_heading***

***procedure***

Name of the procedure that you are declaring or defining.

***procedure_properties***

Each procedure property can appear only once in the procedure declaration. The properties can appear in any order. Properties appear before the `IS` or `AS` keyword in the heading. The properties cannot appear in nested procedures. Only the `ACCESSIBLE BY` property can appear in package procedures.

Standalone procedures may have the following properties in their declaration.

- ACCESSIBLE BY Clause
- DEFAULT COLLATION Clause
- Invoker's Rights and Definer's Rights (AUTHID Property)

***procedure_definition***

Either defines a procedure that was declared earlier or both declares and defines a procedure.

***declare_section***

Declares items that are local to the procedure, can be referenced in *`body`*, and cease to exist when the procedure completes execution.

***body***

Required executable part and optional exception-handling part of the procedure.

**Examples**

- Example 9-1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure"

**Related Topics**

In this chapter:

- "Formal Parameter Declaration"
- "Function Declaration and Definition"

In other chapters:

- "PL/SQL Subprograms"
- "CREATE PROCEDURE Statement"

# Qualified Expression

Using qualified expressions, you can declare and define a complex value in a compact form where the value is needed.

Qualified expressions appear in:

- Collection Variable Declaration
- Expression

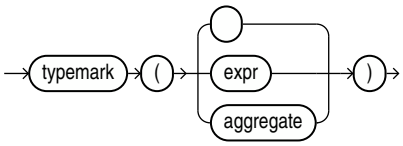
**Topics**

- Syntax
- Semantics

- Examples
- Related Topics

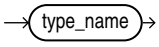
**Syntax**

***qualified_expression* ::=**

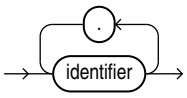



(*typemark* ::=, *aggregate* ::=)

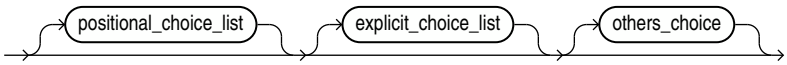
***typemark* ::=**




***type_name* ::=**

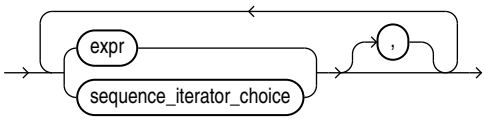



***aggregate* ::=**

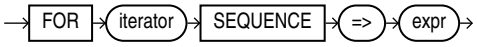



(*positional_choice_list* ::=, *explicit_choice_list* ::= *others_choice* ::=)

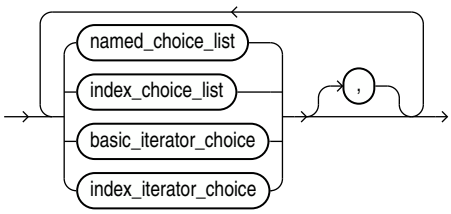
***positional_choice_list* ::=**




***sequence_iterator_choice* ::=**

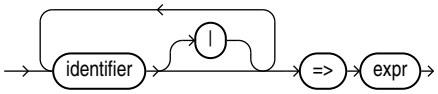


***explicit_choice_list* ::=**

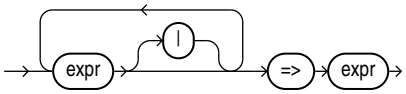



(*named_choice_list* ::=, *indexed_choice_list* ::=, *basic_iterator_choice* ::=, *index_iterator_choice* ::=)

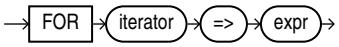
***named_choice_list* ::=**




***indexed_choice_list* ::=**




***basic_iterator_choice* ::=**

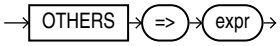



(*iterator* ::=)

***index_iterator_choice* ::=**


(*iterator* ::=)

***others_choice* ::=**



### Semantics

***qualified_expression***

Qualified expressions for `RECORD` types are allowed in any context where an expression of `RECORD` type is allowed.

Qualified expressions for associative array types are allowed in any context where an expression of associative array type is allowed.

***typemark ( aggregate )***

Specifies explicitly the type of the aggregate (qualified items).

***typemark***

Qualified expressions use an explicit type indication to provide the type of the qualified item. This explicit indication is known as a *typemark*.

***type_name***

**[*identifier* . ]*identifier***

Indicates the type of the qualified item.

***aggregate***

A qualified expression combines expression elements to create values of a `RECORD` type, or associative array type.

***positional_choice_list***

***expr* [ , ]**

Positional association is allowed for qualified expressions of `RECORD` type.

A positional association may not follow a named association in the same construct (and vice versa).

***sequence_iterator_choice***

**FOR *iterator* SEQUENCE => *expr***

The sequence iterator choice association is a positional argument and may be intermixed freely with other positional arguments. All positional arguments must precede any non-positional arguments. Sequence iteration is not allowed for INDEX BY VARCHAR2 arrays.

***explicit_choice_list***

***named_choice_list* | *indexed_choice_list* | *basic_iterator_choice* | *index_iterator_choice***

Named choices must use names of fields from the qualifying structure type. Index key values must be compatible with the index type for the qualifying vector type.

***named_choice_list***

A named choice applies only to structured types

***identifier => expr* [ , ]**

Named association is allowed for qualified expressions of `RECORD` type.

***indexed_choice_list***

An index choice applies only to vector types.

***expr => expr* [ , ]**

Indexed choices (key-value pairs) is allowed for qualified expressions of associative array types. Both the key and the value may be expressions.

Using `NULL` as an index key value is not permitted with associative array type constructs.

***basic_iterator_choice***

**FOR *iterator* => *expr***

The basic iterator choice association uses the iterand as an index.

**Restrictions:**

The `PAIRS OF` iteration control may not be used with the basic iterator choice association.

***index_iterator_choice***

**FOR *iterator* INDEX *expr* => *expr***

The index iterator choice association provides an index expression along with the value expression.

***others_choice***

You can use the OTHERS selector in aggregates for record types and aggregates for varrays. The OTHERS choice must be your final choice.

**Examples**

- Assigning Values to RECORD Type Variables Using Qualified Expressions, "Assigning Values to RECORD Type Variables Using Qualified Expressions"
- Example 6-11, "Assigning Values to Associative Array Type Variables Using Qualified Expressions"
- Example 6-8, "Iterator Choice Association in Qualified Expressions"
- Example 6-9, "Index Iterator Choice Association in Qualified Expressions"
- Example 6-10, "Sequence Iterator Choice Association in Qualified Expressions"
- Example 5-27, "Using Dynamic SQL As An Iteration In A Qualified Expression"

**Related Topics**

- "Qualified Expressions Overview" for more conceptual information and examples
- "Expressions"
- Assigning Values to Collection Variables
- Assigning Values to Record Variables

# RAISE Statement

The `RAISE` statement explicitly raises an exception.

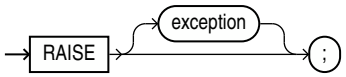
Outside an exception handler, you must specify the exception name. Inside an exception handler, if you omit the exception name, the `RAISE` statement reraises the current exception.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***raise_statement* ::=**




### Semantics

***exception***

Name of an exception, either predefined (see Table 12-3) or user-declared (see "Exception Declaration").

*`exception`* is optional only in an exception handler, where the default is the current exception (see "Reraising Current Exception with RAISE Statement").

### Examples

- Example 12-10, "Declaring, Raising, and Handling User-Defined Exception"
- Example 12-11, "Explicitly Raising Predefined Exception"
- Example 12-12, "Reraising Exception"

### Related Topics

In this chapter:

- "Exception Declaration"
- "Exception Handler"

In other chapters:

- "Raising Exceptions Explicitly"

# Record Variable Declaration

A **record variable** is a composite variable whose internal components, called fields, can have different data types. The value of a record variable and the values of its fields can change.

You reference an entire record variable by its name. You reference a record field with the syntax *record.field*.

You can create a record variable in any of these ways:

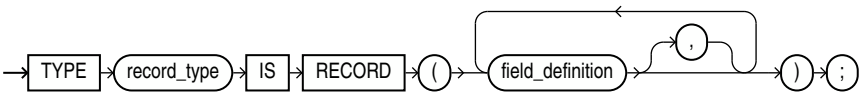
- Define a record type and then declare a variable of that type.
- Use `%ROWTYPE` to declare a record variable that represents either a full or partial row of a database table or view.
- Use `%TYPE` to declare a record variable of the same type as a previously declared record variable.

**Topics**

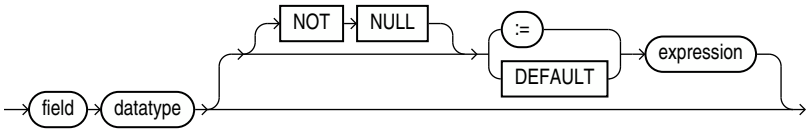

- Syntax
- Semantics
- Examples
- Related Topics


**Syntax**

***record_type_definition* ::=**

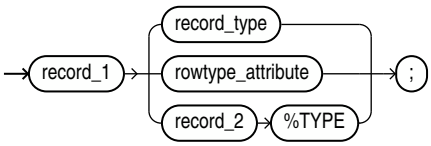



***field_definition* ::=**




( *datatype* ::= , *expression* ::= )

***record_variable_declaration* ::=**



( *rowtype_attribute* ::=)

**Semantics**

***record_type_definition***

***record_type***

Name of the record type that you are defining.

***field_definition***

***field***

Name of the field that you are defining.

***datatype***

Data type of the field that you are defining.

**NOT NULL**

Imposes the `NOT NULL` constraint on the field that you are defining.

For information about this constraint, see "NOT NULL Constraint".

***expression***

Expression whose data type is compatible with *`datatype`*. When *`record_variable_declaration`* is elaborated, the value of *`expression`* is assigned to *`record.field`*. This value is the initial value of the field.

***record_variable_declaration***

***record_1***

Name of the record variable that you are declaring.

***record_type***

Name of a previously defined record type. *`record_type`* is the data type of *`record_1`*.

***rowtype_attribute***

See "%ROWTYPE Attribute".

***record_2***

Name of a previously declared record variable.

**%TYPE**

See "%TYPE Attribute".

**Examples**

- Example 6-41, "RECORD Type Definition and Variable Declaration"
- Example 6-42, "RECORD Type with RECORD Field (Nested Record)"
- Example 6-43, "RECORD Type with Varray Field"

**Related Topics**

In this chapter:

- "Collection Variable Declaration"
- "%ROWTYPE Attribute"

In other chapters:

- "Record Topics"

# RESTRICT_REFERENCES Pragma

The `RESTRICT_REFERENCES` pragma asserts that a user-defined subprogram does not read or write database tables or package variables.

> **Note:**
>
> The `RESTRICT_REFERENCES` pragma is deprecated. Oracle recommends using `DETERMINISTIC` and `PARALLEL_ENABLE` instead of `RESTRICT_REFERENCES`.

Subprograms that read or write database tables or package variables are difficult to optimize, because any invocation of the subprogram might produce different results or encounter errors. If a statement in a user-defined subprogram violates an assertion made by `RESTRICT_REFERENCES`, then the PL/SQL compiler issues an error message when it parses that statement, unless you specify `TRUST`.

Typically, this pragma is specified for functions. If a function invokes procedures, then specify this pragma for those procedures also.

### Restrictions on RESTRICT_REFERENCES Pragma

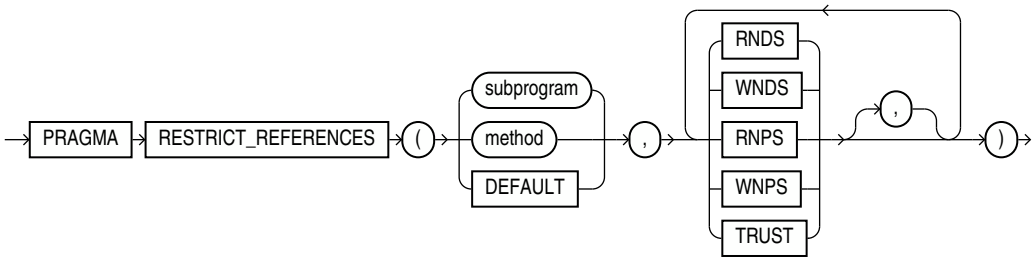
- This pragma can appear only in a package specification or ADT specification.
- Only one `RESTRICT_REFERENCES` pragma can reference a given subprogram.

### Topics

- Syntax
- Semantics

### Syntax

***restrict_references_pragma* ::=**




### Semantics

***subprogram***

Name of a user-defined subprogram, typically a function. If *subprogram* is overloaded, the pragma applies only to the most recent subprogram declaration.

***method***

Name of a MEMBER subprogram. See "CREATE TYPE Statement"for more information.

**DEFAULT**

Applies the pragma to all subprograms in the package specification or ADT specification (including the system-defined constructor for ADTs).

If you also declare the pragma for an individual subprogram, it overrides the DEFAULT pragma for that subprogram.

**RNDS**

Asserts that the subprogram reads no database state (does not query database tables).

**WNDS**

Asserts that the subprogram writes no database state (does not modify tables).

**RNPS**

Asserts that the subprogram reads no package state (does not reference the values of package variables)

**Restriction on RNPS**

You cannot specify RNPS if the subprogram invokes the SQLCODE or SQLERRM function.

**WNPS**

Asserts that the subprogram writes no package state (does not change the values of package variables).

**Restriction on WNPS**

You cannot specify WNPS if the subprogram invokes the SQLCODE or SQLERRM function.

**TRUST**

Asserts that the subprogram can be trusted not to violate the other specified assertions and prevents the PL/SQL compiler from checking the subprogram body for violations. Skipping these checks can improve performance.

If your PL/SQL subprogram invokes a C or Java subprogram, then you must specify TRUST for either the PL/SQL subprogram or the C or Java subprogram, because the PL/SQL compiler cannot check a C or Java subprogram for violations at run time.

> **Note:**
>
> To invoke a subprogram from a parallelized DML statement, you must specify all four constraints—RNDS, WNDS, RNPS, and WNPS. No constraint implies another.

> **See Also:**
>
> *Oracle Database Development Guide* for information about using `PRAGMA RESTRICT_REFERENCES` in existing applications

# RETURN Statement

The `RETURN` statement immediately ends the execution of the subprogram or anonymous block that contains it.

In a function, the `RETURN` statement assigns a specified value to the function identifier and returns control to the invoker, where execution resumes immediately after the invocation (possibly inside the invoking statement). Every execution path in a function must lead to a `RETURN` statement (otherwise, the PL/SQL compiler issues compile-time warning PLW-05005).

In a procedure, the `RETURN` statement returns control to the invoker, where execution resumes immediately after the invocation.

In an anonymous block, the `RETURN` statement exits its own block and all enclosing blocks.

A subprogram or anonymous block can contain multiple `RETURN` statements.

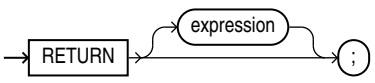
> **Note:**
>
> The `RETURN` statement differs from the `RETURN` clause in a function heading, which specifies the data type of the return value.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***return_statement* ::=**




( *expression* ::= )

**Semantics**

***expression***

Optional when the `RETURN` statement is in a pipelined table function. Required when the `RETURN` statement is in any other function. Not allowed when the `RETURN` statement is in a procedure or anonymous block.

The `RETURN` statement assigns the value of *`expression`* to the function identifier. Therefore, the data type of *`expression`* must be compatible with the data type in the `RETURN` clause of the function. For information about expressions, see "Expression".

**Examples**

- Example 9-3, "Execution Resumes After RETURN Statement in Function"
- Example 9-4, "Function Where Not Every Execution Path Leads to RETURN Statement"
- Example 9-5, "Function Where Every Execution Path Leads to RETURN Statement"
- Example 9-6, "Execution Resumes After RETURN Statement in Procedure"
- Example 9-7, "Execution Resumes After RETURN Statement in Anonymous Block"

**Related Topics**

In this chapter:

- "Block"
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"

In other chapters:

- "RETURN Statement"

# RETURNING INTO Clause

The `RETURNING INTO` clause specifies the variables in which to store the values returned by the statement to which the clause belongs.

The variables can be either individual variables or collections. If the statement affects no rows, then the values of the variables are undefined.

The **static** `RETURNING INTO` clause belongs to a `DELETE`, `INSERT`, `UPDATE`, or `MERGE` statement. The **dynamic** `RETURNING INTO` clause belongs to the `EXECUTE IMMEDIATE` statement.

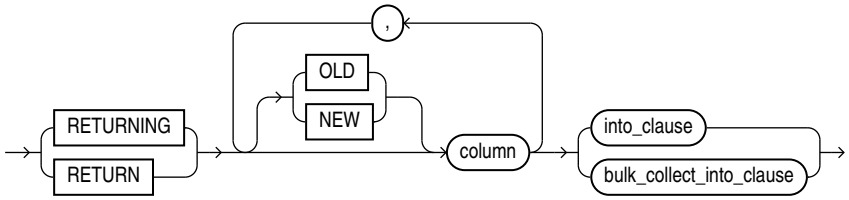
> **Note:**
>
> You cannot use the `RETURNING INTO` clause for remote or parallel deletes.

**Topics**

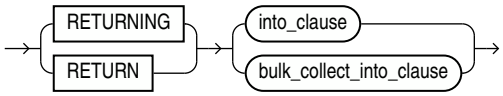
- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

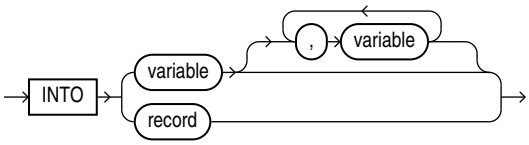
***static_returning_clause* ::=**



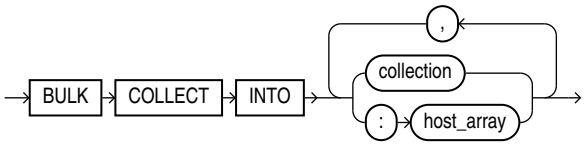

***dynamic_returning_clause* ::=**




***into_clause* ::=**




***bulk_collect_into_clause* ::=**




### Semantics

***static_returning_clause***

***OLD | NEW***

Given columns `c1` and `c2` in a table, you can specify `OLD` for column `c1` (for example `OLD c1`). You can also specify `OLD` for a column referenced by a column expression (for example `c1+OLD c2`) or on a column referenced by an aggregate function (for example `AVG(OLD c1)`). When `OLD` is specified for a column, the column value before the execution of an associated `INSERT`, `UPDATE`, `MERGE`, or `DELETE` statement is returned. In the case of a column referenced by a column expression, what is returned is the result from evaluating the column expression using the column value before the DML statement is executed.

`NEW` can be explicitly specified for a column, a column referenced in an expression, or a column referenced by an aggregate function to return a column value after the `INSERT`, `UPDATE`, `MERGE`, or `DELETE` statement, or an expression result that uses the after execution value of a column.

When `OLD` and `NEW` are both omitted for a column or an expression, the post DML execution column value (pre-execution value for `DELETE`), or the expression result computed using the column values after DML execution, is returned.

Note that it is valid to specify `OLD` and `NEW` on constants (for example, `OLD 1`), however, the keywords are ignored. `OLD` and `NEW` are not currently supported on virtual columns.

> **Note:**
>
> While `UPDATE` statements have both before and after update column values, `INSERT` statements have no `OLD` column value and `DELETE` statements have no `NEW` column value. Although using `OLD` and `NEW` in these cases is valid, nothing is returned.

***column***

Expression whose value is the name of a column of a database table.

***into_clause***

Specifies the variables or record in which to store the column values that the statement returns.

**Restriction on *into_clause***

Use *`into_clause`* in *`dynamic_returning_clause`* if and only if *`dynamic_sql_stmt`* (which appears in "EXECUTE IMMEDIATE Statement") returns a single row.

***record***

The name of a record variable in which to store the row that the statement returns. For each *select_list* item in the statement, the record must have a corresponding, type-compatible field.

***variable***

Either the name of a scalar variable in which to store a column that the statement returns or the name of a host cursor variable that is declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Each *select_list* item in the statement must have a corresponding, type-compatible variable. The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

***bulk_collect_into_clause***

Specifies one or more existing collections or host arrays in which to store the rows that the statement returns. For each *select_list* item in the statement, *`bulk_collect_into_clause`* must have a corresponding, type-compatible *`collection`* or *`host_array`*.

For the reason to use this clause, see "Bulk SQL and Bulk Binding".

**Restrictions on *bulk_collect_into_clause***

- Use the *`bulk_collect_into_clause`* clause in *`dynamic_returning_clause`* if and only if *`dynamic_sql_stmt`* (which appears in "EXECUTE IMMEDIATE Statement") can return multiple rows.

- You cannot use *bulk_collect_into_clause* in client programs.
- When the statement that includes *bulk_collect_into_clause* requires implicit data type conversions, *bulk_collect_into_clause* can have only one *collection* or *host_array*.

***collection***

Name of a collection variable in which to store the rows that the statement returns.

**Restrictions on *collection***

- *collection* cannot be the name of an associative array that is indexed by a string.
- When the statement requires implicit data type conversions, *collection* cannot be the name of a collection of a composite type.

**:*host_array***

Name of an array declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *host_array*.

**Examples**

- Example 6-58, "UPDATE Statement Assigns Values to Record Variable"
- Example 7-1, "Static SQL Statements"
- Example 13-25, "Returning Deleted Rows in Two Nested Tables"
- Example 13-26, "Returning NEW and OLD Values of Updated Rows"
- Example 13-27, "DELETE with RETURN BULK COLLECT INTO in FORALL Statement"

**Related Topics**

In this chapter:

- "DELETE Statement Extension"
- "EXECUTE IMMEDIATE Statement"
- "FETCH Statement"
- "SELECT INTO Statement"
- "UPDATE Statement Extensions"

In other chapters:

- "Using SQL Statements to Return Rows in PL/SQL Record Variables"
- "EXECUTE IMMEDIATE Statement"
- "RETURNING INTO Clause with BULK COLLECT Clause"

# RESULT_CACHE Clause

Indicates to store the function results into the server result cache.

The RESULT_CACHE clause can appear in the following SQL statements:

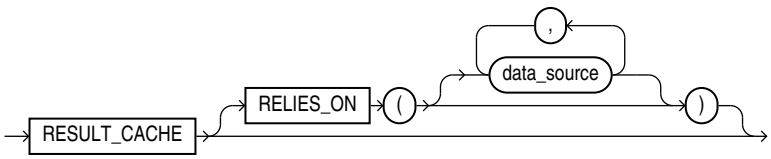
- CREATE FUNCTION Statement
- CREATE TYPE BODY Statement

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***result_cache_clause* ::=**




### Semantics

***result_cache_clause***

*result_cache_clause* can appear only once in the function.

**RESULT_CACHE**

To make a function result-cached, include the RESULT_CACHE clause in the function definition. If you declare the function before defining it, you must also include the RESULT_CACHE option in the function declaration.

**Restriction on RESULT_CACHE**

- `RESULT_CACHE` is disallowed on functions with `OUT` or `IN OUT` parameters.
- `RESULT_CACHE` is disallowed on functions with `IN` or `RETURN` parameter of (or containing) these types:
  - `BLOB`
  - `CLOB`
  - `NCLOB`
  - `REF CURSOR`
  - Collection
  - Object
  - Record or PL/SQL collection that contains an unsupported return type
- `RESULT_CACHE` is disallowed on function in an anonymous block.
- `RESULT_CACHE` is disallowed on pipelined table function, nested function and SQL macro.

**RELIES_ON**

Specifies the data sources on which the results of the function depend. Each *`data_source`* is the name of either a database table or view.

> **Note:**
>
> - This clause is deprecated. As of Oracle Database 12c, the database detects all data sources that are queried while a result-cached function is running, and `RELIES_ON` clause does nothing.
> - You cannot use `RELIES_ON` clause in a function declared in an anonymous block.

**Examples**

- Examples of Result-Cached Functions

**Related Topics**

In this chapter:

- Function Declaration and Definition

In other chapters:

- PL/SQL Function Result Cache

In other books:

- *Oracle Database Performance Tuning Guide*

# %ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record that represents either a full or partial row of a database table or view.

For every visible column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly. Making an invisible column visible changes the structure of some records declared with the `%ROWTYPE` attribute.

The record fields do not inherit the constraints or initial values of the corresponding columns.

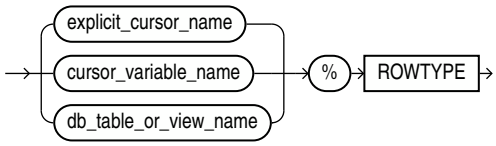
The `%ROWTYPE` attribute cannot be used if the referenced character column has a collation other than `USING_NLS_COMP`.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***rowtype_attribute* ::=**




### Semantics

***rowtype_attribute***

***explicit_cursor_name***

Name of an explicit cursor. For every column selected by the query associated with *explicit_cursor_name*, the record has a field with the same name and data type.

***cursor_variable_name***

Name of a strong cursor variable. For every column selected by the query associated with *cursor_variable_name*, the record has a field with the same name and data type.

***db_table_or_view_name***

Name of a database table or view that is accessible when the declaration is elaborated. For every column of *db_table_or_view_name*, the record has a field with the same name and data type.

### Examples

- Example 6-45, "%ROWTYPE Variable Represents Full Database Table Row"
- Example 6-46, "%ROWTYPE Variable Does Not Inherit Initial Values or Constraints"
- Example 6-47, "%ROWTYPE Variable Represents Partial Database Table Row"
- Example 6-48, "%ROWTYPE Variable Represents Join Row"
- Example 6-51, "%ROWTYPE Affected by Making Invisible Column Visible"
- Example 6-54, "Assigning %ROWTYPE Record to RECORD Type Record"

### Related Topics

In this chapter:

- "Cursor Variable Declaration"
- "Explicit Cursor Declaration and Definition"
- "Record Variable Declaration"
- "%TYPE Attribute"

In other chapters:

- About Data-Bound Collation
- "Declaring Items using the %ROWTYPE Attribute"

- "%ROWTYPE Attribute and Invisible Columns"

# Scalar Variable Declaration

A scalar variable stores a value with no internal components. The value can change. A scalar variable declaration specifies the name and data type of the variable and allocates storage for it.

The declaration can also assign an initial value and impose the `NOT NULL` constraint.

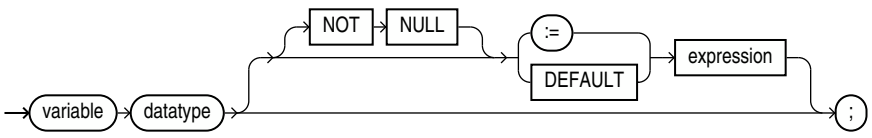
You reference a scalar variable by its name.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***variable_declaration* ::=**




( *expression* ::= )

**Semantics**

***variable_declaration***

***variable***

Name of the variable that you are declaring.

***datatype***

Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics.

For information about scalar data types, see "PL/SQL Data Types".

**NOT NULL**

Imposes the `NOT NULL` constraint on the variable. For information about this constraint, see "NOT NULL Constraint".

***expression***

Value to be assigned to the variable when the declaration is elaborated. *`expression`* and *`variable`* must have compatible data types.

**Examples**

- Example 3-11, "Scalar Variable Declarations"
- Example 3-13, "Variable and Constant Declarations with Initial Values"
- Example 3-14, "Variable Initialized to NULL by Default"
- Example 3-9, "Variable Declaration with NOT NULL Constraint"

**Related Topics**

In this chapter:

- "Assignment Statement"
- "Collection Variable Declaration"
- "Constant Declaration"
- "Expression"
- "Record Variable Declaration"
- "%ROWTYPE Attribute"
- "%TYPE Attribute"

In other chapters:

- "Declaring Variables"
- "PL/SQL Data Types"

# SELECT INTO Statement

The `SELECT INTO` statement retrieves values from one or more database tables (as the SQL `SELECT` statement does) and stores them in variables (which the SQL `SELECT` statement does not do).

> **Caution:**
>
> The `SELECT INTO` statement with the `BULK COLLECT` clause is vulnerable to aliasing, which can cause unexpected results. For details, see "SELECT BULK COLLECT INTO Statements and Aliasing".

> **See Also:**
>
> *Oracle Database SQL Language Reference* for the syntax of the SQL `SELECT` statement

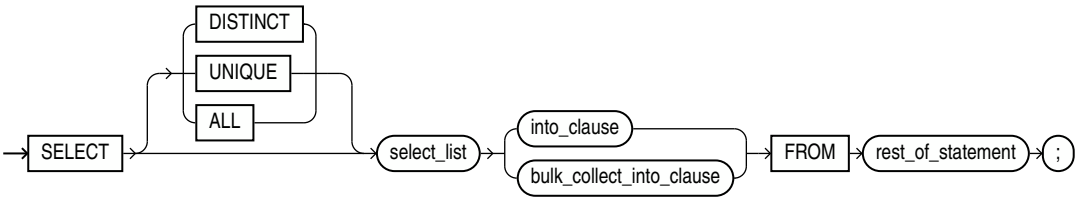
**Topics**

- Syntax
- Semantics

- Examples
- Related Topics

### Syntax

***select_into_statement* ::=**




( *bulk_collect_into_clause* ::=, *into_clause* ::=, *Oracle Database SQL Language Reference* for *select_list* syntax)

### Semantics

***select_into_statement***

**DISTINCT | UNIQUE**

Specify `DISTINCT` or `UNIQUE` if you want the database to return only one copy of each set of duplicate rows selected. These two keywords are synonymous. Duplicate rows are those with matching values for each expression in the select list.

**Restrictions on DISTINCT and UNIQUE Queries**

- The total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter `DB_BLOCK_SIZE`.
- You cannot specify `DISTINCT` if the *select_list* contains LOB columns.

**ALL**

**(Default)** Causes the database to return all rows selected, including all copies of duplicates.

***select_list***

If the `SELECT INTO` statement returns no rows, PL/SQL raises the predefined exception `NO_DATA_FOUND`. To guard against this exception, select the result of the aggregate function `COUNT(*)`, which returns a single value even if no rows match the condition.

***into_clause***

With this clause, the `SELECT INTO` statement retrieves one or more columns from a single row and stores them in either one or more scalar variables or one record variable. For more information, see "*into_clause* ::=".

***bulk_collect_into_clause***

With this clause, the `SELECT INTO` statement retrieves an entire result set and stores it in one or more collection variables. For more information, see "*bulk_collect_into_clause* ::=".

***rest_of_statement***

Anything that can follow the keyword `FROM` in a SQL `SELECT` statement, described in *Oracle Database SQL Language Reference*.

**Examples**

- Example 3-25, "Assigning Value to Variable with SELECT INTO Statement"
- Example 6-56, "SELECT INTO Assigns Values to Record Variable"
- Example 7-37, "ROLLBACK Statement"
- Example 7-38, "SAVEPOINT and ROLLBACK Statements"
- Example 7-43, "Declaring Autonomous Function in Package"
- Example 8-20, "Validation Checks Guarding Against SQL Injection"
- Example 13-16, "Bulk-Selecting Two Database Columns into Two Nested Tables"
- Example 13-17, "Bulk-Selecting into Nested Table of Records"
- Example 13-21, "Limiting Bulk Selection with ROWNUM, SAMPLE, and FETCH FIRST"

**Related Topics**

In this chapter:

- "Assignment Statement"
- "FETCH Statement"
- "%ROWTYPE Attribute"

In other chapters:

- "Assigning Values to Variables with the SELECT INTO Statement"
- "Using SELECT INTO to Assign a Row to a Record Variable"
- "Processing Query Result Sets With SELECT INTO Statements"
- "SELECT INTO Statement with BULK COLLECT Clause"

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about the SQL `SELECT` statement

# SERIALLY_REUSABLE Pragma

The `SERIALLY_REUSABLE` pragma specifies that the package state is needed for only one call to the server (for example, an OCI call to the database or a stored procedure invocation through a database link).

After this call, the storage for the package variables can be reused, reducing the memory overhead for long-running sessions.

This pragma is appropriate for packages that declare large temporary work areas that are used once in the same session.

The SERIALLY_REUSABLE pragma can appear in the *declare_section* of the specification of a bodiless package, or in both the specification and body of a package, but not in only the body of a package.

**Topics**

- Syntax
- Examples
- Related Topics

**Syntax**

***serially_reusable_pragma* ::=**


**Examples**

- Example 11-4, "Creating SERIALLY_REUSABLE Packages"
- Example 11-5, "Effect of SERIALLY_REUSABLE Pragma"
- Example 11-6, "Cursor in SERIALLY_REUSABLE Package Open at Call Boundary"

**Related Topics**

- "SERIALLY_REUSABLE Packages"
- "Pragmas"

# SHARD_ENABLE Clause

The SHARD_ENABLE keyword indicates that a query referencing the defined function can be pushed down into the shards of a sharded database (SDB).

When using the SHARD_ENABLE clause, the query optimizer takes the initiative to push the execution of the PL/SQL function to the shards.

The SHARD_ENABLE clause can appear in the following SQL statement:

- CREATE FUNCTION Statement
- CREATE PACKAGE Statement

Queries with PL/SQL functions created **with** the SHARD_ENABLE keyword will be pushed down, if possible, to the shards and executed as multishard queries. *If possible* refers to the fact that there may be other parts of the query that do not allow the pushdown. Therefore, the optimizer will make the pushdown decision.

Queries with PL/SQL functions created **without** the SHARD_ENABLE keyword will not be pushed down to the shards and executed as cross shard queries on the coordinator.

**Topics**

- Syntax
- Semantics

- Usage Notes
- Related Topics

### Syntax

***shard_enable_clause* ::=**


### Semantics

***shard_enable_clause***

### Usage Notes

It is up to you to decide whether a function execution can be pushed to the shards. However, there are some instances where you should decide not to use `shard_enable`:

- Functions referencing any session context variables that may be different on the shards and coordinator.
- Functions referencing any global variables that may be different on the shards and coordinator.
- Functions referencing any data local to the coordinator.

In some cases you may decide it is safe to push a function, even if it references a package global variable or reads data from a table.

Even if a PL/SQL function is marked with `SHARD_ENABLE` clause, there are times when the evaluation needs to happen on the coordinator, meaning the function evaluation is not pushed to shards. Possible scenarios include:

- When the function is in `SELECT` list and there is a join between sharded tables and the join is not on a sharding key (note that a join between sharded and duplicated key is okay),
- When the function is in `SELECT` list and there is a join with a local (non-sharded) table,
- If such a function is present in `WHERE` clause and it takes input parameters as column of multiple sharded tables and there is no join on sharding key.

Pushing eligible functions down to shards to execute as multi shard queries rather than cross shard queries can result in significant performance improvement by:

- Distributing the computation by performing evaluation of PL/SQL functions on each shard.
- Reducing the size of the data returned from shards when the predicate involves a PL/SQL function, resulting in smaller inputs for joins on the coordinator.

### Related Topics

In other books:

- *Oracle Globally Distributed Database Guide* for more information about sharding in the Oracle Database

# SHARING Clause

The `SHARING` clause applies only when creating an object in an application root in the context of an application maintenance. This type of object is called an application common object and it can be shared with the application PDBs that belong to the application root.

The `SHARING` clause can appear in the context of creating application common PL/SQL and SQL objects with these SQL statements.

**Table 14-2 Summary of Possible Sharing Attributes by Application Common Object Type**

| Application Common Object Statement | Possible SHARING Attributes | Syntax and Semantics |
|---|---|---|
| CREATE ANALYTIC VIEW | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE ATTRIBUTE DIMENSION | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE CLUSTER | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE CONTEXT | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE DIRECTORY | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE FUNCTION | METADATA, NONE | CREATE FUNCTION Statement |
| CREATE HIERARCHY | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE INDEXTYPE | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE JAVA | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE LIBRARY | METADATA, NONE | CREATE LIBRARY Statement |
| CREATE MATERIALIZED VIEW LOG | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE OPERATOR | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE PROCEDURE | METADATA, NONE | CREATE PROCEDURE Statement |
| CREATE PACKAGE | METADATA, NONE | CREATE PACKAGE Statement |
| CREATE PACKAGE BODY | METADATA, NONE | CREATE PACKAGE BODY Statement |
| CREATE SEQUENCE | METADATA, DATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE SYNONYM | METADATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE TABLE | METADATA, DATA, EXTENDED DATA, NONE | *Oracle Database SQL Language Reference* |
| CREATE TRIGGER | METADATA, NONE | CREATE TRIGGER Statement |
| CREATE TYPE | METADATA, NONE | CREATE TYPE Statement |
| CREATE TYPE BODY | METADATA, NONE | CREATE TYPE BODY Statement |

**Table 14-2 (Cont.) Summary of Possible Sharing Attributes by Application Common Object Type**

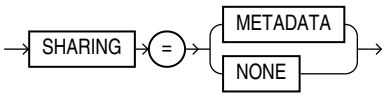
| Application Common Object Statement | Possible SHARING Attributes | Syntax and Semantics |
|---|---|---|
| CREATE VIEW | METADATA, DATA, EXTENDED DATA, NONE | *Oracle Database SQL Language Reference* |

### Topics

- Syntax
- Semantics
- Related Topics

### Syntax

***sharing_clause* ::=**




### Semantics

#### *sharing_clause*

Specifies how the object is shared using one of the following sharing attributes:

- **`METADATA`** - A metadata link shares the metadata, but its data is unique to each container. This type of object is referred to as a **metadata-linked application common object**.
- **`NONE`** - The object is not shared and can only be accessed in the application root.

If you omit this clause during an application operation, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the object. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

#### Restrictions on `SHARING` clause

The sharing clause may only appear during an application installation, upgrade or patch in an application root. You must issue an `ALTER PLUGGABLE DATABASE APPLICATION ... BEGIN` statement to start the operation and an `ALTER PLUGGABLE DATABASE APPLICATION ... END` statement to end the operation. The *sharing_clause* is illegal outside this context and this implies the object is not shared.

You cannot change the sharing attribute of an object after it is created.

Generally, common objects cannot depend on local objects. The exceptions to this rule are :

- the common object being created is a synonym
- the common object being created depends on a local operator

If you try to create a common object that depends on local objects other than these two exceptions, it will result in the creation of a local object.

**Related Topics**

In other books:

- *Oracle Database Concepts* for more information about application maintenance
- *Oracle Database Concepts* for an example of patching an application using the automated technique
- *Oracle Database Reference* for more information on the `DEFAULT_SHARING` initialization parameter
- *Oracle Database Reference* for more information on the `ALL_OBJECTS` view initialization `SHARING` and `APPLICATION` columns
- *Oracle Database Administrator's Guide* for complete information on creating application common objects

# SQL_MACRO Clause

The `SQL_MACRO` clause marks a function as a SQL macro which can be used as either a scalar expression or a table expression.

A `TABLE` macro is a function annotated as a `SQL_MACRO` and defined as a `TABLE` type.

A `SCALAR` macro is a function annotated as a `SQL_MACRO` and defined as a `SCALAR` type.

A SQL macro referenced in a view is always processed with the view owner's privileges.

The `AUTHID` property cannot be specified. When a SQL macro is invoked, the function body executes with definer's rights to construct the text to return. The resulting expression is evaluated with invoker's rights. The SQL macro owner must grant inherit privileges to the invoking function.

When a macro annotated function is used in PL/SQL, it works like a regular function returning character or `CLOB` type with no macro expansion.

Many `SCALAR` macros can instead be written as standard PL/SQL functions, which can be called directly from a SQL statement. The PL/SQL function is automatically converted into a semantically equivalent SQL expression by the SQL Transpiler. This converted SQL expression is used during execution, replacing the call to the original PL/SQL function.

Transpilation can improve performance by removing the need to switch between the SQL runtime and the PL/SQL runtime. Where possible, transpilation is performed automatically on any PL/SQL function called from a SQL statement (unless the feature has been explicitly disabled).

For more information about the SQL Transpiler, see *Oracle Database SQL Tuning Guide*.

The SQL_MACRO annotation can appear in the following SQL statement:

- CREATE FUNCTION Statement

**SQL Macro Usage Restrictions:**

- A `TABLE` macro can only appear in `FROM` clause of a query table expression.
- A `SCALAR` macro cannot appear in `FROM` clause of a query table expression. It can appear wherever PL/SQL functions are allowed, for example in the select list, the `WHERE` clause, and the `ORDER BY` clause.
- A scalar macro cannot have table arguments.

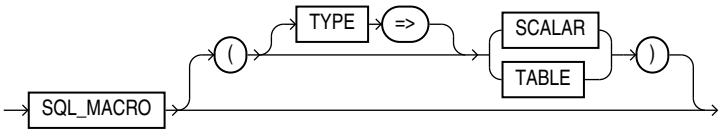
- A SQL macro cannot appear in a virtual column expression, functional index, editioning view or materialized view.
- Type methods cannot be annotated with `SQL_MACRO`.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***sql_macro_clause* ::=**




### Semantics

#### *sql_macro_clause*

The *sql_macro_clause* can appear only once in the function. To make a SQL macro function, include the *sql_macro_clause* in the function definition. If you declare the SQL macro function before defining it, you must specify the *sql_macro_clause* in the function declaration.

If `SCALAR` or `TABLE` is not specified, `TABLE` is the default.

#### SCALAR

Specify `SCALAR` if the macro function can be used in scalar expressions.

**TABLE** (Default)

Specify `TABLE` if the macro function can be used in table expressions.

#### Restrictions on *sql_macro_clause*

The `SQL_MACRO` annotation is disallowed with `RESULT_CACHE`, `PARALLEL_ENABLE`, and `PIPELINED`. Although the `DETERMINISTIC` property cannot be specified, a SQL macro is always implicitly deterministic.

The SQL macro function must have a return type of `VARCHAR2`, `CHAR`, or `CLOB`.

### Examples

#### Example 14-36 Emp_doc: Using a Scalar Macro to Convert Columns into a JSON or XML Document

The emp_doc SQL macro converts employee fields into a document string (JSON or XML).

```
The macro is implemented as a tree of nested macros with the following call graph
structure.
emp_doc()
   ==> emp_json()
```

```
          ==> name_string()
          ==> email_string()
                ==> name_string()
          ==> date_string()
   ==> emp_xml
          ==> name_string()
          ==> email_string()
                ==> name_string()
          ==> date_string()
```

The date_string function converts a date in a string formatted as a four digits year, month (01-12) and day of the month (1-31).

```
CREATE FUNCTION date_string(dat DATE)
                    RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
   RETURN q'{
             TO_CHAR(dat, 'YYYY-MM-DD')
          }';
END;
/
```

The name_string function sets the first letter of each words in the first_name and last_name in uppercase and all other letters in lowercase. It concatenates the formatted first name with a space and the formatted last name, and removes leading and trailing spaces from the resulting string.

```
CREATE FUNCTION name_string(first_name VARCHAR2,
                                last_name VARCHAR2)
                     RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
   RETURN q'{
          TRIM(INITCAP(first_name) || ' ' || INITCAP(last_name))
          }';
END;
/
```

The email_string sets the email address using the name_string function with the first_name and last_name and replacing all spaces with a period, and appending a default domain name of example.com.

```
CREATE FUNCTION email_string(first_name VARCHAR2,
                                last_name VARCHAR2,
                                host_name VARCHAR2 DEFAULT 'example.com')
                    RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
   RETURN q'{
          REPLACE(LOWER(name_string(first_name, last_name)),' ','.') || '@' || host_name
          }';
END;
/
```

The emp_json SQL macro returns a JSON document string.

```
CREATE FUNCTION emp_json(first_name VARCHAR2 DEFAULT NULL,
                             last_name VARCHAR2 DEFAULT NULL,
                             hire_date DATE DEFAULT NULL,
                             phone_num VARCHAR2 DEFAULT NULL)
                    RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
   RETURN q'{
          JSON_OBJECT(
             'name'      : name_string(first_name, last_name),
```

```
            'email'     : email_string(first_name, last_name),
            'phone'     : phone_num,
            'hire_date' : date_string(hire_date)
            ABSENT ON NULL)
         }';
END;
/
```

The emp_xml SQL macro returns an XML document string.

```
CREATE FUNCTION emp_xml(first_name VARCHAR2 DEFAULT NULL,
                        last_name VARCHAR2 DEFAULT NULL,
                        hire_date DATE DEFAULT NULL,
                        phone_num VARCHAR2 DEFAULT NULL)
                  RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
   RETURN q'{
       XMLELEMENT("xml",
                  CASE WHEN first_name || last_name IS NOT NULL THEN
                     XMLELEMENT("name", name_string(first_name, last_name))
                  END,
                  CASE WHEN first_name || last_name IS NOT NULL THEN
                     XMLELEMENT("email", email_string(first_name, last_name))
                  END,
                  CASE WHEN hire_date IS NOT NULL THEN
                     XMLELEMENT("hire_date", date_string(hire_date))
                  END,
                  CASE WHEN phone_num IS NOT NULL THEN
                     XMLELEMENT("phone", phone_num)
                  END)
          }';
END;
/
```

The emp_doc SQL macro returns employee fields into a JSON (default) or XML document string.

```
CREATE FUNCTION emp_doc(first_name VARCHAR2 DEFAULT NULL,
                        last_name VARCHAR2 DEFAULT NULL,
                        hire_date DATE DEFAULT NULL,
                        phone_num VARCHAR2 DEFAULT NULL,
                        doc_type VARCHAR2 DEFAULT 'json')
                  RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
   RETURN q'{
     DECODE(LOWER(doc_type),
            'json', emp_json(first_name, last_name, hire_date, phone_num),
            'xml', emp_xml(first_name, last_name, hire_date, phone_num))
        }';
END;
/
```

This query shows the emp_doc SQL macro used in a scalar expression to list all employees in a JSON document string in department 30.

```
SELECT department_id,
           emp_doc(first_name => e.first_name, hire_date => e.hire_date) doc
FROM hr.employees e
WHERE department_id = 30
ORDER BY last_name;
```

Result:

```
     30
{"name":"Shelli","email":"shelli@example.com","hire_date":"2015-12-24"}
     30 {"name":"Karen","email":"karen@example.com","hire_date":"2017-08-10"}
     30 {"name":"Guy","email":"guy@example.com","hire_date":"2016-11-15"}
     30
{"name":"Alexander","email":"alexander@example.com","hire_date":"2013-05-18"}
     30 {"name":"Den","email":"den@example.com","hire_date":"2012-12-07"}
     30 {"name":"Sigal","email":"sigal@example.com","hire_date":"2015-07-24"}
```

This query shows the emp_doc SQL macro used in a scalar expression to list all employees in a XML document string.

```
SELECT deptno,
           emp_doc(first_name => ename, hire_date => hiredate, doc_type => 'xml') doc
FROM scott.emp
ORDER BY ename;
```

Result:

```
20 <xml><name>Adams</name><email>adams@example.com</email><hire_date>1987-05-23</
hire_date></xml>
30 <xml><name>Allen</name><email>allen@example.com</email><hire_date>1981-02-20</
hire_date></xml>
30 <xml><name>Blake</name><email>blake@example.com</email><hire_date>1981-05-01</
hire_date></xml>
10 <xml><name>Clark</name><email>clark@example.com</email><hire_date>1981-06-09</
hire_date></xml>
20 <xml><name>Ford</name><email>ford@example.com</email><hire_date>1981-12-03</
hire_date></xml>
...
30 <xml><name>Ward</name><email>ward@example.com</email><hire_date>1981-02-22</
hire_date></xml>

VARIABLE surname VARCHAR2(100)
EXEC :surname := 'ellison'
WITH e AS (SELECT emp.*, :surname lname FROM emp WHERE deptno IN (10,20))
SELECT deptno,
       emp_doc(first_name => ename, last_name => lname, hire_date => hiredate) doc
FROM e
ORDER BY ename;
```

Result:

```
10 {"name":"Clark Ellison","email":"clark.ellison@example.com","hire_date":"1981-06-09"}
20 {"name":"Ford Ellison","email":"ford.ellison@example.com","hire_date":"1981-12-03"}
20 {"name":"Jones Ellison","email":"jones.ellison@example.com","hire_date":"1981-04-02"}
10 {"name":"King Ellison","email":"king.ellison@example.com","hire_date":"1981-11-17"}
10 {"name":"Miller
Ellison","email":"miller.ellison@example.com","hire_date":"1982-01-23"}
20 {"name":"Scott Ellison","email":"scott.ellison@example.com","hire_date":"1987-04-19"}
20 {"name":"Smith Ellison","email":"smith.ellison@example.com","hire_date":"1980-12-17"}
```

### Example 14-37 Env: Using a Scalar Macro in a Scalar Expression

The env SQL macro provides a wrapper for the value of the parameter associated with the context namespace USERENV which describes the current session.

```
CREATE PACKAGE env AS
   FUNCTION current_user RETURN VARCHAR2 SQL_MACRO(SCALAR);
   FUNCTION current_edition_name RETURN VARCHAR2 SQL_MACRO(SCALAR);
   FUNCTION module RETURN VARCHAR2 SQL_MACRO(SCALAR);
   FUNCTION action RETURN VARCHAR2 SQL_MACRO(SCALAR);
END;
/
CREATE PACKAGE BODY env AS
   FUNCTION current_user RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
     BEGIN
        RETURN q'{SYS_CONTEXT('userenv','SESSION_USER')}';
     END;
   FUNCTION current_edition_name RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
     BEGIN
        RETURN q'{SYS_CONTEXT('userenv','CURRENT_EDITION_NAME')}';
     END;
   FUNCTION module RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
     BEGIN
        RETURN q'{SYS_CONTEXT('userenv','MODULE')}';
     END;
   FUNCTION action RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
     BEGIN
        RETURN q'{SYS_CONTEXT('userenv','ACTION')}';
     END;
END;
/
```

Select the current user info.

```
SELECT env.current_user, env.module, env.action FROM DUAL;
```

Result:

```
SCOTT   SQL*PLUS
```

### Example 14-38 Budget : Using a Table Macro in a Table Expression

This example shows the SQL macro named budget used in a table expression to return the total salary in each department for employees for a given job title.

```
CREATE FUNCTION budget(job VARCHAR2) RETURN VARCHAR2 SQL_MACRO IS
BEGIN
   RETURN q'{SELECT deptno, SUM(sal) budget
             FROM scott.emp
             WHERE job = budget.job
             GROUP BY deptno}';
END;
```

This query shows the SQL macro budget used in a table expression.

```
SELECT * FROM budget('MANAGER');
```

Result:

```
DEPTNO     BUDGET
---------- ----------
        20       2975
        30       2850
        10       2450
```

### Example 14-39 Take: Using a Table Macro with a Polymorphic View

This example creates a table macro named take which returns the first n rows from table t.

```
CREATE FUNCTION take (n NUMBER, t DBMS_TF.table_t)
                      RETURN VARCHAR2 SQL_MACRO IS
BEGIN
   RETURN 'SELECT * FROM t FETCH FIRST take.n ROWS ONLY';
END;
/
```

The query returns the first two rows from table dept.

```
SELECT * FROM take(2, dept);
```

Result:

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
```

```
VAR row_count NUMBER
EXEC :row_count := 5

WITH t AS (SELECT * FROM emp NATURAL JOIN dept ORDER BY ename)
SELECT ename, dname FROM take(:row_count, t);
```

Result:

```
ENAME      DNAME
---------- --------------
ADAMS      RESEARCH
ALLEN      SALES
BLAKE      SALES
CLARK      ACCOUNTING
FORD       RESEARCH
```

### Example 14-40 Range : Using a Table Macro in a Table Expression

This example creates a SQL macro that generates an arithmetic progression of rows in the range [first, stop]. The first row start with the value *first*, and each subsequent row's value will be *step* more than the previous row's value.

The following combination of arguments will produce zero rows:

- step < 0 and first < stop
- step = 0
- step > 0 and first > stop

```
/*  PACKAGE NAME: GEN
 *  SQL TABLE MACROS:
 *     range(stop  : number to generate starting from zero)
 *     range(first : starting number of the sequence (default=0),
 *           stop  : generate numbers up to, but not including this number,
 *           step  : difference between each number in the sequence (default=1) )
*/
CREATE PACKAGE gen IS
   FUNCTION range(stop NUMBER)
            RETURN VARCHAR2 SQL_MACRO(TABLE);
```

```
   FUNCTION range(first NUMBER DEFAULT 0, stop NUMBER, step NUMBER DEFAULT 1)
            RETURN VARCHAR2 SQL_MACRO(TABLE);

   FUNCTION tab(tab TABLE, replication_factor NATURAL)
            RETURN TABLE PIPELINED ROW POLYMORPHIC USING gen;

   FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T, replication_factor NATURAL)
            RETURN DBMS_TF.DESCRIBE_T;

   PROCEDURE fetch_rows(replication_factor NATURALN);
END gen;
/
CREATE PACKAGE BODY gen IS
   FUNCTION range(stop NUMBER)
            RETURN VARCHAR2 SQL_MACRO(TABLE) IS
   BEGIN
      RETURN q'{SELECT ROWNUM-1 n FROM gen.tab(DUAL, stop)}';
   END;

   FUNCTION range(first NUMBER DEFAULT 0, stop NUMBER, step NUMBER DEFAULT 1)
           RETURN VARCHAR2 SQL_MACRO(TABLE) IS
   BEGIN
      RETURN q'{
             SELECT first+n*step n FROM gen.range(ROUND((stop-first)/NULLIF(step,0)))
             }';
   END;

   FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T, replication_factor NATURAL)
            RETURN DBMS_TF.DESCRIBE_T AS
   BEGIN
      RETURN DBMS_TF.DESCRIBE_T(row_replication => true);
   END;

  PROCEDURE fetch_rows(replication_factor NATURALN) as
  BEGIN
    DBMS_TF.ROW_REPLICATION(replication_factor);
  END;
END gen;
/
```

The gen.get_range SQL macro is used in table expressions.

This query returns a sequence of 5 rows starting at zero.

```
SELECT * FROM gen.range(5);
```

Result:

```
         0
         1
         2
         3
         4
```

This query returns a sequence starting at 5, stopping at 10 (not included).

```
SELECT * FROM gen.range(5, 10);
```

Result:

```
5
6
7
8
9
```

This query returns a sequence starting at 0, stopping at 1, by increment of 0.1.

```
SELECT * FROM gen.range(0, 1, step=>0.1);
```

Result:

```
0
.1
.2
.3
.4
.5
.6
.7
.8
.9
```

This query returns a sequence starting at 5, stopping at -6 (not included) by decrement of 2.

```
SELECT * FROM gen.range(+5,-6,-2);
```

Result:

```
5
3
1
-1
-3
-5
```

**Related Topics**

- Overview of Polymorphic Table Functions
- *Oracle Database PL/SQL Packages and Types Reference* for more information about how to specify the PTF implementation package and use the `DBMS_TF` utilities
- *Oracle Database Reference* for more information about the SQL_MACRO column in the ALL_PROCEDURES view

# SQLCODE Function

In an exception handler, the `SQLCODE` function returns the numeric code of the exception being handled. (Outside an exception handler, `SQLCODE` returns `0`.)

For an internally defined exception, the numeric code is the number of the associated Oracle Database error. This number is negative except for the error "no data found", whose numeric code is +100.

For a user-defined exception, the numeric code is either +1 **(default)** or the error code associated with the exception by the `EXCEPTION_INIT` pragma.

A SQL statement cannot invoke `SQLCODE`.

If a function invokes `SQLCODE`, and you use the `RESTRICT_REFERENCES` pragma to assert the purity of the function, then you cannot specify the constraints `WNPS` and `RNPS`.

**Topics**

- Syntax
- Examples
- Related Topics

**Syntax**

***sqlcode_function* ::=**


**Examples**

- Example 12-23, "Displaying SQLCODE and SQLERRM Values"

**Related Topics**

In this chapter:

- "Block"
- "EXCEPTION_INIT Pragma"
- "Exception Handler"
- "RESTRICT_REFERENCES Pragma"
- "SQLERRM Function"

In other chapters:

- "Retrieving Error Code and Error Message"

> **See Also:**
>
> *Oracle Database Error Messages Reference* for a list of Oracle Database error messages and information about them, including their numbers

# SQLERRM Function

The `SQLERRM` function returns the error message associated with an error code.

> **Note:**
>
> The language of the error message depends on the `NLS_LANGUAGE` parameter. For information about this parameter, see *Oracle Database Globalization Support Guide*.

A SQL statement cannot invoke `SQLERRM`.

If a function invokes `SQLERRM`, and you use the `RESTRICT_REFERENCES` pragma to assert the purity of the function, then you cannot specify the constraints `WNPS` and `RNPS`.

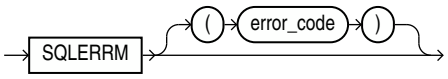
> **Note:**
>
> `DBMS_UTILITY.FORMAT_ERROR_STACK` is recommended over `SQLERRM`, unless you use the `FORALL` statement with its `SAVE EXCEPTIONS` clause. For more information, see "Retrieving Error Code and Error Message".

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***sqlerrm_function* ::=**




**Semantics**

***sqlerrm_function***

***error_code***

Expression whose value is an Oracle Database error code.

**Default:** error code associated with the current value of `SQLCODE`.

Like `SQLCODE`, `SQLERRM` without *`error_code`* is useful only in an exception handler. Outside an exception handler, or if the value of *`error_code`* is zero, `SQLERRM` returns ORA-0000.

If the value of *`error_code`* is +100, `SQLERRM` returns ORA-01403.

If the value of *`error_code`* is a positive number other than +100, `SQLERRM` returns this message:

```
-error_code: non-ORACLE exception
```

If the value of *`error_code`* is a negative number whose absolute value is an Oracle Database error code, `SQLERRM` returns the error message associated with that error code. For example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-6511): ' || TO_CHAR(SQLERRM(-6511)));
END;
/
```

Result:

```
SQLERRM(-6511): ORA-06511: PL/SQL: cursor already open
```

If the value of *error_code* is a negative number whose absolute value is not an Oracle Database error code, SQLERRM returns this message:

```
ORA-error_code: Message error_code not found; product=RDBMS;
facility=ORA
```

For example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-50000): ' || TO_CHAR(SQLERRM(-50000)));
END;
/
```

Result:

```
SQLERRM(-50000): ORA-50000: Message 50000 not found; product=RDBMS;
facility=ORA
```

### Examples

- Example 12-23, "Displaying SQLCODE and SQLERRM Values"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"

### Related Topics

In this chapter:

- "Block"
- "EXCEPTION_INIT Pragma"
- "RESTRICT_REFERENCES Pragma"
- "SQLCODE Function"

In other chapters:

- "Retrieving Error Code and Error Message"

> **See Also:**
>
> *Oracle Database Error Messages Reference* for a list of Oracle Database error messages and information about them

# SUPPRESSES_WARNING_6009 Pragma

The SUPPRESSES_WARNING_6009 pragma marks a subroutine to indicate that the PLW-06009 warning is suppressed at its call site in an OTHERS exception handler. The marked subroutine has the same effect as a RAISE statement and suppresses the PLW-06009 compiler warning.

The OTHERS exception handler does not issue the compiler warning PLW-06009 if an exception is raised explicitly using either a RAISE statement or the RAISE_APPLICATION_ERROR procedure as the last statement. Similarly, a call to a subroutine marked with the

SUPPRESSES_WARNING_6009 pragma, from the OTHERS exception handler, does not issue the PLW-06009 warning.

The SUPPRESSES_WARNING_6009 pragma can appear in the following SQL statements :

- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement
- CREATE PROCEDURE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***suppresses_warning_6009_pragma ::=***


**Semantics**

***suppresses_warning_6009_pragma***

The SUPPRESSES_WARNING_6009 pragma applies to standalone subprograms, packaged subprograms and to the methods in an Abstract Data Type definition.

For a standalone subprogram, the SUPPRESSES_WARNING_6009 pragma may appear as the first item in the declaration block of a subprogram definition, immediately after the keyword IS or AS.

In a package specification, a package body and a type specification, the SUPPRESSES_WARNING_6009 pragma must appear immediately after the subprogram declaration.

If the subprogram has separate declaration and definition, the SUPPRESSES_WARNING_6009 pragma may be applied either to the subprogram declaration, or to the subprogram definition, or to both.

For overloaded subprograms, SUPPRESSES_WARNING_6009 pragma only applies to the marked overload.

When the SUPPRESSES_WARNING_6009 pragma is applied to a subprogram in a package specification, the subprogram is marked for use both in the package body and in the invokers of the package.

When the SUPPRESSES_WARNING_6009 pragma is applied to a subprogram in the definition of a package body, the subprogram is marked for use only in the package body even if the subprogram is declared in the package specification.

The SUPPRESSES_WARNING_6009 pragma applied to a subprogram in a base type object, is inherited in a derived type object unless there is an override without the pragma in the derived type object.

The SUPPRESSES_WARNING_6009 pragma may be terminated with a "," when applied to a subprogram in a type specification. In all other contexts, the pragma is terminated by ";".

The SUPPRESSES_WARNING_6009 pragma on the subprogram provides a hint to the compiler to suppress the warning PLW-06009 at its call site.

***pls_identifier***

Identifier of the PL/SQL element to which the pragma is applied.

The identifier is a parameter to the SUPPRESSES_WARNING_6009 pragma that must name the subprogram to which it is applied.

If the identifier in the SUPPRESSES_WARNING_6009 does not identify a subroutine in the declaration section, the pragma has no effect.

### Examples

#### Example 14-41 Enabling the PLW-6009 Warning

This example shows how to set the PLSQL_WARNINGS parameter to enable the PLW-6009 warning in a session for demonstration purpose.

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:(6009)';
```

#### Example 14-42 SUPPRESSES_WARNING_6009 Pragma in a Procedure

This example shows a standalone procedure p1 marked with the SUPPRESSES_WARNING_6009 pragma. The p1 procedure is invoked from an OTHERS exception handler in procedure p2, which does not raise an exception explicitly.

```
CREATE PROCEDURE p1
AUTHID DEFINER
IS
    PRAGMA SUPPRESSES_WARNING_6009(p1);
BEGIN
    RAISE_APPLICATION_ERROR(-20000, 'Unexpected error raised');
END;
/
```

The PLW-06009 warning is not issued when compiling procedure p2.

```
CREATE PROCEDURE p2
AUTHID DEFINER
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('In procedure p2');
EXCEPTION
  WHEN OTHERS THEN
```

```
    p1;
END p2;
/
```

### Example 14-43 SUPPRESSES_WARNING_6009 Pragma in a Function

This example shows a standalone function f1 marked with the `SUPPRESSES_WARNING_6009` pragma. This function is invoked from an `OTHERS` exception handler in function f2, which does not explicitly have a `RAISE` statement.

```
CREATE FUNCTION f1(id NUMBER) RETURN NUMBER
AUTHID DEFINER
IS
  PRAGMA SUPPRESSES_WARNING_6009(f1);
  x NUMBER;
BEGIN
  x := id + 1;
RETURN x;
END;
/
```

The `PLW-06009` warning is not issued when compiling function f2.

```
CREATE FUNCTION f2(numval NUMBER) RETURN NUMBER
AUTHID DEFINER
IS
  i NUMBER;
BEGIN
  i := numval + 1;
  RETURN i;
EXCEPTION
  WHEN OTHERS THEN
    RETURN f1(i);
END;
/
```

### Example 14-44 SUPPRESSES_WARNING_6009 Pragma in an Overloaded Subprogram in a Package Specification

This example shows an overloaded procedure p1, declared in a package specification. Only the second overload of p1 is marked with the `SUPPRESSES_WARNING_6009` pragma. This marked overload is invoked from the `OTHERS` exception handler in procedure p6, which does not have an explicit `RAISE` statement.

```
CREATE PACKAGE pk1 IS
   PROCEDURE p1(x NUMBER);
   PROCEDURE p1;
   PRAGMA SUPPRESSES_WARNING_6009(p1);
```

```
END;
/



CREATE OR REPLACE PACKAGE BODY pk1 IS
   PROCEDURE p1(x NUMBER) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('In the first overload');
   END;

   PROCEDURE p1 IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('In the second overload');
      RAISE_APPLICATION_ERROR(-20000, 'Unexpected error');
   END;
END;
/
```

The procedure p6 invokes the p1 second overloaded procedure. The compiler does not issue a `PLW-06009` warning when compiling procedure p6.

```
CREATE OR REPLACE PROCEDURE p6 AUTHID DEFINER IS
j NUMBER := 5;
BEGIN
   j := j + 2;
EXCEPTION
   WHEN OTHERS THEN
       pk1.p1;
END;
/
```

### Example 14-45 SUPPRESSES_WARNING_6009 Pragma in a Forward Declaration in a Package Body

This example shows a forward declaration subprogram marked with the `SUPPRESSES_WARNING_6009` pragma in a package body. This marked procedure pn is invoked from the `OTHERS` exception handler in procedure p5, which has no `RAISE` statement.

```
CREATE OR REPLACE PACKAGE pk2 IS
   PROCEDURE p5;
END;
/
```

The compiler does not issue a `PLW-06009` warning when creating the package body.

```
CREATE OR REPLACE PACKAGE BODY pk2 IS
   PROCEDURE pn; /* Forward declaration */
   PRAGMA SUPPRESSES_WARNING_6009(pn);

   PROCEDURE p5 IS
```

```
   BEGIN
      DBMS_OUTPUT.PUT_LINE('Computing');
   EXCEPTION
      WHEN OTHERS THEN
         pn;
   END;

   PROCEDURE pn IS
   BEGIN
      RAISE_APPLICATION_ERROR(-20000, 'Unexpected error');
   END;
END;
/
```

### Example 14-46 SUPPRESSES_WARNING_6009 Pragma in Object Type Methods

This example shows the `SUPPRESSES_WARNING_6009` pragma applied to a member method declared in the newid Abstract Data Type (ADT) definition. The marked procedure log_error is invoked from the `OTHERS` exception handler in the type body, which does not have a `RAISE` statement.

```
CREATE OR REPLACE TYPE newid AUTHID DEFINER
AS OBJECT(
  ID1 NUMBER,
  MEMBER PROCEDURE incr,
  MEMBER PROCEDURE log_error,
  PRAGMA SUPPRESSES_WARNING_6009(log_error)
);
/
```

The compiler does not issue the `PLW-06009` warning when compiling the type body.

```
CREATE OR REPLACE TYPE BODY newid
AS
  MEMBER PROCEDURE incr
  IS
  BEGIN
     DBMS_OUTPUT.PUT_LINE('Computing value');
  EXCEPTION
     WHEN OTHERS THEN
        log_error;
  END;

  MEMBER PROCEDURE log_error
  IS
  BEGIN
    RAISE_APPLICATION_ERROR(-20000, 'Unexpected error');
  END;
END;
/
```

#### Related Topics

- Exception Handler

- Pragmas
- PL/SQL Error Handling
- RAISE Statement
- Raising Exceptions Explicitly

# %TYPE Attribute

The `%TYPE` attribute lets you declare a constant, variable, collection element, record field, or subprogram parameter to be of the same data type as a previously declared variable or column (without knowing what that type is).

The `%TYPE` attribute cannot be used if the referenced character column has a collation other than `USING_NLS_COMP`.

The item declared with `%TYPE` is the **referencing item**, and the previously declared item is the **referenced item**.

The referencing item inherits the following from the referenced item:

- Data type and size
- Constraints (unless the referenced item is a column)

The referencing item does not inherit the initial value of the referenced item.

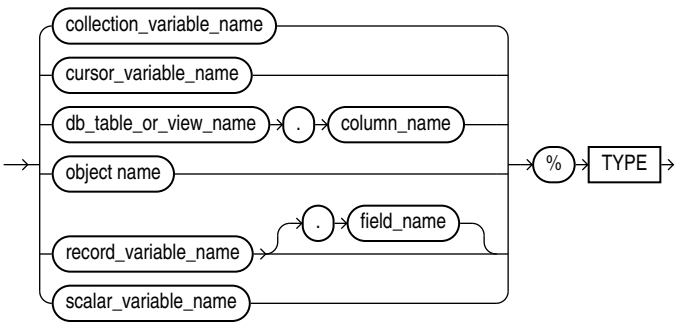
If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***type_attribute* ::=**



### Semantics

***type_attribute***

***collection_variable_name***

Name of a collection variable.

**Restriction on *collection_variable_name***

In a constant declaration, *collection_variable* cannot be an associative array variable.

***cursor_variable_name***

Name of a cursor variable.

***db_table_or_view_name***

Name of a database table or view that is accessible when the declaration is elaborated.

***column_name***

Name of a column of *db_table_or_view*.

***object_name***

Name of an instance of an ADT.

***record_variable_name***

Name of a record variable.

***field_name***

Name of a field of *record_variable*.

***scalar_variable_name***

Name of a scalar variable.

### Examples

- Example 3-15, "Declaring Variable of Same Type as Column"
- Example 3-16, "Declaring Variable of Same Type as Another Variable"

### Related Topics

In this chapter:

- "Constant Declaration"
- "%ROWTYPE Attribute"
- "Scalar Variable Declaration"

In other chapters:

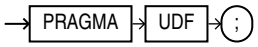
- About Data-Bound Collation
- "Declaring Items using the %TYPE Attribute"

# UDF Pragma

The `UDF` pragma tells the compiler that the PL/SQL unit is a **user defined function** that is used primarily in SQL statements, which might improve its performance.

**Syntax**

***udf_pragma* ::=**




# UPDATE Statement Extensions

PL/SQL extends the *`update_set_clause`* and *`where_clause`* of the SQL `UPDATE` statement as follows:

- In the *`update_set_clause`*, you can specify a record. For each selected row, the `UPDATE` statement updates each column with the value of the corresponding record field.
- In the *`where_clause`*, you can specify a `CURRENT OF` clause, which restricts the `UPDATE` statement to the current row of the specified cursor.

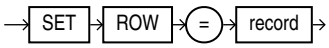
> **See Also:**
>
> *Oracle Database SQL Language Reference* for the syntax of the SQL `UPDATE` statement

**Topics**

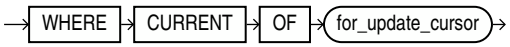
- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

***update_set_clause* ::=**




***where_clause* ::=**



### Semantics

***update_set_clause***

***record***

Name of a record variable that represents a row of the item described by *dml_table_expression_clause*. That is, for every column of the row, the record must have a field with a compatible data type. If a column has a `NOT NULL` constraint, then its corresponding field cannot have a `NULL` value.

***where_clause***

***for_update_cursor***

Name of a `FOR UPDATE` cursor; that is, an explicit cursor associated with a `SELECT FOR UPDATE` statement.

### Examples

- Example 6-61, "Updating Rows with Records"

### Related Topics

In this chapter:

- "Explicit Cursor Declaration and Definition"
- "Record Variable Declaration"
- "%ROWTYPE Attribute"

In other chapters:

- "Updating Rows with Records"
- "Restrictions on Record Inserts and Updates"
- "SELECT FOR UPDATE and FOR UPDATE Cursors"

# WHILE LOOP Statement

The `WHILE LOOP` statement runs one or more statements while a condition is `TRUE`.

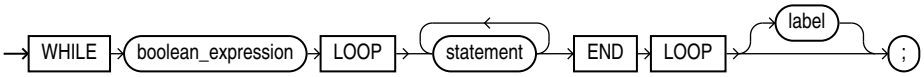
The `WHILE LOOP` statement ends when the condition becomes `FALSE` or `NULL`, or when a statement inside the loop transfers control outside the loop or raises an exception.

### Topics

- Syntax
- Semantics
- Examples
- Related Topics

### Syntax

***while_loop_statement* ::=**




( *boolean_expression* ::= , *statement* ::= )

### Semantics

***while_loop_statement***

***boolean_expression***

Expression whose value is `TRUE`, `FALSE`, or `NULL`.

*`boolean_expression`* is evaluated at the beginning of each iteration of the loop. If its value is `TRUE`, the statements after `LOOP` run. Otherwise, control transfers to the statement after the `WHILE LOOP` statement.

***statement***

To prevent an infinite loop, at least one statement must change the value of *`boolean_expression`* to `FALSE` or `NULL`, transfer control outside the loop, or raise an exception. The statements that can transfer control outside the loop are:

- "CONTINUE Statement" (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement"
- "GOTO Statement"
- "RAISE Statement"

***label***

Label that identifies *`while_loop_statement`* (see "*statement* ::=" and "*label*"). `CONTINUE`, `EXIT`, and `GOTO` statements can reference this label.

Labels improve readability, especially when `LOOP` statements are nested, but only if you ensure that the label in the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

### Examples

#### Example 14-47 WHILE LOOP Statements

The statements in the first `WHILE LOOP` statement never run, and the statements in the second `WHILE LOOP` statement run once.

```
DECLARE
  done  BOOLEAN := FALSE;
BEGIN
  WHILE done LOOP
    DBMS_OUTPUT.PUT_LINE ('This line does not print.');
    done := TRUE;  -- This assignment is not made.
  END LOOP;
```

```
  WHILE NOT done LOOP
    DBMS_OUTPUT.PUT_LINE ('Hello, world!');
    done := TRUE;
  END LOOP;
END;
/
```

Result:

```
Hello, world!
```

### Related Topics

- "Basic LOOP Statement"
- "Cursor FOR LOOP Statement"
- "Explicit Cursor Declaration and Definition"
- "FETCH Statement"
- "FOR LOOP Statement"
- "FORALL Statement"
- "OPEN Statement"
- "WHILE LOOP Statement"