15
SQL Statements for Stored PL/SQL Units

This chapter explains how to use the SQL statements that create, change, and drop stored PL/SQL units.

CREATE [OR REPLACE] Statements

Each of these SQL statements creates a PL/SQL unit at schema level and stores it in the database:

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

Each of these `CREATE` statements has an optional `OR REPLACE` clause. Specify `OR REPLACE` to re-create an existing PL/SQL unit—that is, to change its declaration or definition without dropping it, re-creating it, and regranting object privileges previously granted on it. If you redefine a PL/SQL unit, the database recompiles it.

> **Caution:**
>
> A `CREATE OR REPLACE` statement does not issue a warning before replacing the existing PL/SQL unit.

None of these `CREATE` statements can appear in a PL/SQL block.

ALTER Statements

To recompile an existing PL/SQL unit without re-creating it (without changing its declaration or definition), use one of these SQL statements:

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

Reasons to use an `ALTER` statement are:

- To explicitly recompile a stored unit that has become invalid, thus eliminating the need for implicit runtime recompilation and preventing associated runtime compilation errors and performance overhead.
- To recompile a stored unit with different compilation parameters.
- To enable or disable a trigger.
- To specify the `EDITIONABLE` or `NONEDITIONABLE` property of a stored unit whose schema object type is not yet editionable in its schema.

The `ALTER TYPE` statement has additional uses.

DROP Statements

To drop an existing PL/SQL unit from the database, use one of these SQL statements:

- DROP FUNCTION Statement
- DROP LIBRARY Statement
- DROP PACKAGE Statement
- DROP PROCEDURE Statement
- DROP TRIGGER Statement
- DROP TYPE Statement
- DROP TYPE BODY Statement

Related Topics

- For instructions for reading the syntax diagrams in this chapter, see *Oracle Database SQL Language Reference*.
- For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.
- For information about compilation parameters, see "PL/SQL Units and Compilation Parameters".

ALTER FUNCTION Statement

The `ALTER FUNCTION` statement explicitly recompiles a standalone function.

Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

Note:

This statement does not change the declaration or definition of an existing function. To redeclare or redefine a standalone function, use the "CREATE FUNCTION Statement" with the `OR REPLACE` clause.

Topics

- Prerequisites
- Syntax

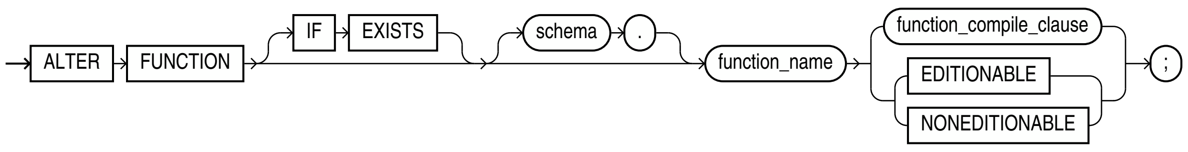
- Semantics
- Example
- Related Topics

Prerequisites

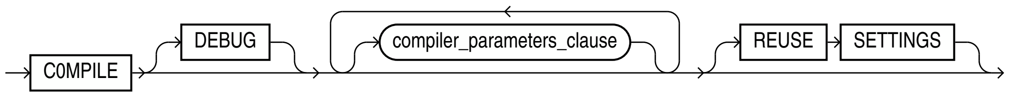
If the function is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the function must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

alter_function **::=**



function_compile_clause **::=**



(*compiler_parameters_clause* ::=)

Semantics

alter_function

IF EXISTS

Recompiles the function if it exists. If no such function exists, the statement is ignored without error.

schema

Name of the schema containing the function. **Default:** your schema.

function_name

Name of the function to be recompiled.

{ EDITIONABLE | NONEDITIONABLE }

Specifies whether the function becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `FUNCTION` in *`schema`*. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

function_compile_clause

Recompiles the function, whether it is valid or invalid.

See *compile_clause* semantics.

See also DEFAULT COLLATION Clause compilation semantics.

Example

Example 15-1 Recompiling a Function

To explicitly recompile the function `get_bal` owned by the sample user `oe`, issue this statement:

```
ALTER FUNCTION oe.get_bal COMPILE;
```

If the database encounters no compilation errors while recompiling `get_bal`, then `get_bal` becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling `get_bal` results in compilation errors, then the database returns an error, and `get_bal` remains invalid.

The database also invalidates all objects that depend upon `get_bal`. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

Related Topics

- "CREATE FUNCTION Statement"
- "DROP FUNCTION Statement"

ALTER LIBRARY Statement

The `ALTER LIBRARY` statement explicitly recompiles a library.

Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

Note:

This statement does not change the declaration or definition of an existing library. To redeclare or redefine a library, use the "CREATE LIBRARY Statement" with the `OR REPLACE` clause.

Topics

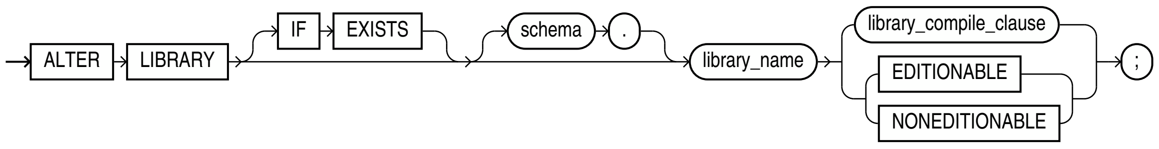
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

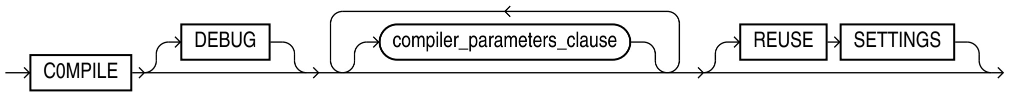
If the library is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the library must be in your schema or you must have the `ALTER ANY LIBRARY` system privilege.

Syntax

alter_library **::=**



library_compile_clause **::=**



(*compiler_parameters_clause* ::=)

Semantics

alter_library

IF EXISTS

Alters the library if it exists. If no such library exists, the statement is ignored without error.

library_name

Name of the library to be altered.

{ EDITIONABLE | NONEDITIONABLE }

Specifies whether the library becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `LIBRARY` in *`schema`*. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

library_compile_clause

Recompiles the library.

See *compile_clause* and *compiler_parameters_clause* semantics.

Example

Example 15-2 Recompiling a Library

To explicitly recompile the library `my_ext_lib` owned by the sample user `hr`, issue this statement:

```
ALTER LIBRARY IF EXISTS hr.my_ext_lib COMPILE;
```

If the database encounters no compilation errors while recompiling `my_ext_lib`, then `my_ext_lib` becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling `my_ext_lib` results in compilation errors, then the database returns an error, and `my_ext_lib` remains invalid.

The database also invalidates all objects that depend upon `my_ext_lib`. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

If `my_ext_lib` does not already exist in the schema, this statement is ignored without error due to the `IF EXISTS` clause. Note that the output message is the same whether or not the library exists (in this case, `Library altered`).

Related Topics

- "CREATE LIBRARY Statement"
- "DROP LIBRARY Statement"

ALTER PACKAGE Statement

The `ALTER PACKAGE` statement explicitly recompiles a package specification, body, or both. Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the `ALTER PACKAGE` statement recompiles all package objects. You cannot use the `ALTER PROCEDURE` statement or `ALTER FUNCTION` statement to recompile individually a procedure or function that is part of a package.

Note:

This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the "CREATE PACKAGE Statement", or the "CREATE PACKAGE BODY Statement" with the `OR REPLACE` clause.

Topics

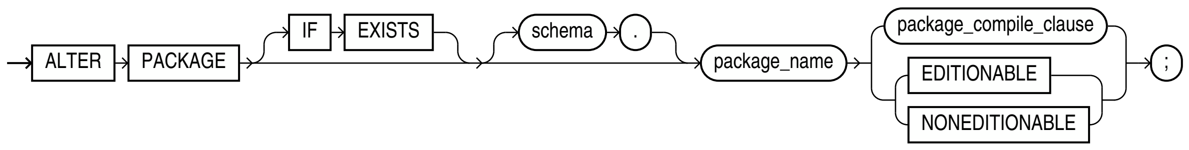
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

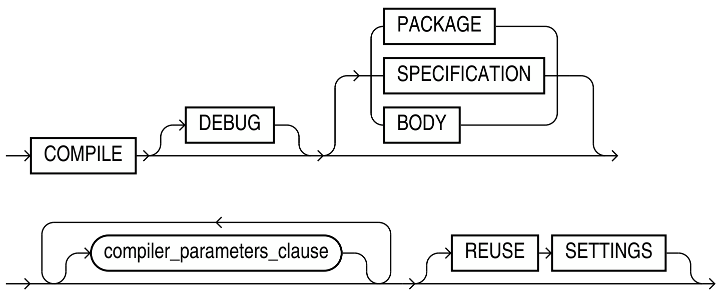
If the package is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the package must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

alter_package **::=**



package_compile_clause **::=**



(*compiler_parameters_clause* ::=)

Semantics

alter_package

IF EXISTS

Recompiles the package if it exists. If no such package exists, the statement is ignored without error.

schema

Name of the schema containing the package. **Default:** your schema.

package_name

Name of the package to be recompiled.

{ EDITIONABLE | NONEDITIONABLE }

Specifies whether the package becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `PACKAGE` in *`schema`*. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

package_compile_clause

Recompiles the package specification, body, or both.

See *compile_clause* and *compiler_parameters_clause* semantics.

Examples

Example 15-3 Recompiling a Package

This statement explicitly recompiles the specification and body of the `hr.emp_mgmt` package.

See "CREATE PACKAGE Statement" for the example that creates this package.

```
ALTER PACKAGE emp_mgmt COMPILE PACKAGE;
```

If the database encounters no compilation errors while recompiling the `emp_mgmt` specification and body, then `emp_mgmt` becomes valid. The user `hr` can subsequently invoke or reference all package objects declared in the specification of `emp_mgmt` without runtime recompilation. If recompiling `emp_mgmt` results in compilation errors, then the database returns an error and `emp_mgmt` remains invalid.

The database also invalidates all objects that depend upon `emp_mgmt`. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

To recompile the body of the `emp_mgmt` package in the schema `hr`, issue this statement:

```
ALTER PACKAGE hr.emp_mgmt COMPILE BODY;
```

If the database encounters no compilation errors while recompiling the package body, then the body becomes valid. The user `hr` can subsequently invoke or reference all package objects declared in the specification of `emp_mgmt` without runtime recompilation. If recompiling the body results in compilation errors, then the database returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of `emp_mgmt`, the database does not invalidate dependent objects.

Related Topics

- "CREATE PACKAGE Statement"
- "DROP PACKAGE Statement"

ALTER PROCEDURE Statement

The `ALTER PROCEDURE` statement explicitly recompiles a standalone procedure.

Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the "ALTER PACKAGE Statement").

Note:

This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a standalone procedure, use the "CREATE PROCEDURE Statement" with the `OR REPLACE` clause.

The `ALTER PROCEDURE` statement is very similar to the `ALTER FUNCTION` statement. See "ALTER FUNCTION Statement" for more information.

Topics

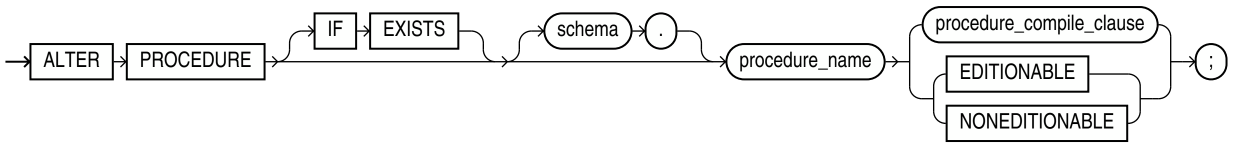
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

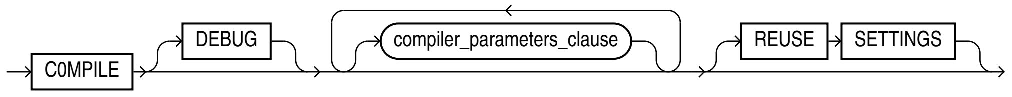
If the procedure is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the procedure must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

alter_procedure **::=**



procedure_compile_clause **::=**



(*compiler_parameters_clause* ::=)

Semantics

alter_procedure

IF EXISTS

Alters the procedure if it exists. If no such procedure exists, the statement is ignored without error.

schema

Name of the schema containing the procedure. **Default:** your schema.

procedure_name

Name of the procedure to be altered.

{ EDITIONABLE | NONEDITIONABLE }

Specifies whether the procedure becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `PROCEDURE` in *`schema`*. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

procedure_compile_clause

See *compile_clause* and *compiler_parameters_clause* semantics.

Example

Example 15-4 Recompiling a Procedure

To explicitly recompile the procedure `remove_emp` owned by the user `hr`, issue this statement:

```
ALTER PROCEDURE IF EXISTS hr.remove_emp COMPILE;
```

If the database encounters no compilation errors while recompiling `remove_emp`, then `remove_emp` becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling `remove_emp` results in compilation errors, then the database returns an error and `remove_emp` remains invalid.

The database also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that invoke `remove_emp`. If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

If `remove_emp` does not already exist in the schema, this statement is ignored without error due to the `IF EXISTS` clause. Note that the output message is the same whether or not the procedure exists (in this case, `Procedure altered`).

Related Topics

- "CREATE PROCEDURE Statement"
- "DROP PROCEDURE Statement"

ALTER TRIGGER Statement

The `ALTER TRIGGER` statement enables, disables, compiles, or renames a database trigger.

Note:

This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the "CREATE TRIGGER Statement" with the `OR REPLACE` clause.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

If the trigger is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the trigger must be in your schema or you must have `ALTER ANY TRIGGER` system privilege.

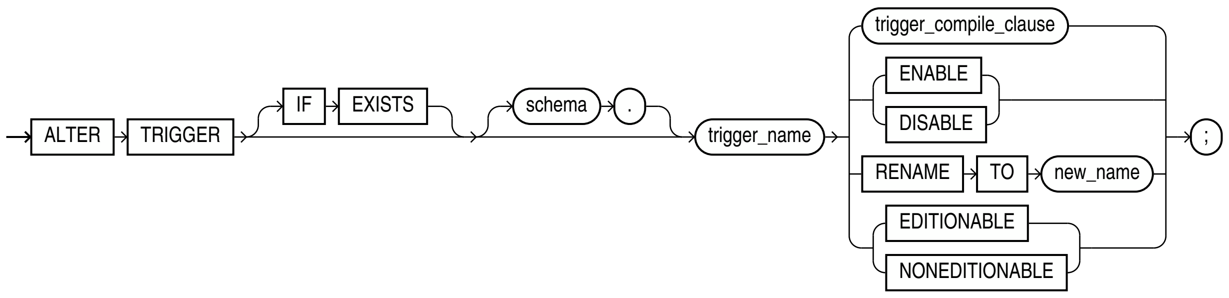
In addition, to alter a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege.

See Also:

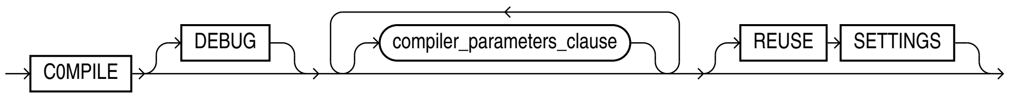
"CREATE TRIGGER Statement" for more information about triggers based on `DATABASE` triggers

Syntax

alter_trigger **::=**



trigger_compile_clause **::=**



(*compiler_parameters_clause* ::=)

Semantics

alter_trigger

IF EXISTS

Enables, disables, compiles, or renames the trigger if it exists. If no such trigger exists, the statement is ignored without error.

schema

Name of the schema containing the trigger. **Default:** your schema.

trigger_name

Name of the trigger to be altered.

[ENABLE | DISABLE]

Enables or disables the trigger.

RENAME TO *new_name*

Renames the trigger without changing its state.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the `*_SOURCE` static data dictionary views. As a result, comments and formatting may change in the `TEXT` column of those views even though the trigger source did not change.

{ EDITIONABLE | NONEDITIONABLE }

Specifies whether the trigger becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `TRIGGER` in *`schema`*. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

Restriction on NONEDITIONABLE

You cannot specify `NONEDITIONABLE` for a crossedition trigger.

trigger_compile_clause

Recompiles the trigger, whether it is valid or invalid.

See *compile_clause* and *compiler_parameters_clause* semantics.

Examples

Example 15-5   Disabling Triggers

The sample schema `hr` has a trigger named `update_job_history` created on the `employees` table. The trigger fires whenever an `UPDATE` statement changes an employee's `job_id`. The trigger inserts into the `job_history` table a row that contains the employee's ID, begin and end date of the last job, and the job ID and department.

When this trigger is created, the database enables it automatically. You can subsequently disable the trigger with this statement:

```
ALTER TRIGGER update_job_history DISABLE;
```

When the trigger is disabled, the database does not fire the trigger when an `UPDATE` statement changes an employee's job.

Example 15-6   Enabling Triggers

After disabling the trigger, you can subsequently enable it with this statement:

```
ALTER TRIGGER update_job_history ENABLE;
```

After you reenable the trigger, the database fires the trigger whenever an `UPDATE` statement changes an employee's job. If an employee's job is updated while the trigger is disabled, then the database does not automatically fire the trigger for this employee until another transaction changes the `job_id` again.

Related Topics

In this chapter:

- "CREATE TRIGGER Statement"
- "DROP TRIGGER Statement"

In other chapters:

- "Trigger Compilation, Invalidation, and Recompilation"

- "Trigger Enabling and Disabling"

ALTER TYPE Statement

Use the ALTER TYPE statement to add or drop member attributes or methods. You can change the existing properties of an object type, and you can modify the scalar attributes of the type. You can also use this statement to recompile the specification or body of the type or to change the specification of an object type by adding new object member subprogram specifications.

The `ALTER TYPE` statement does one of the following to a type that was created with "CREATE TYPE Statement" and "CREATE TYPE BODY Statement":

- **Evolves** the type; that is, adds or drops member attributes or methods.

  For more information about type evolution, see *Oracle Database Object-Relational Developer's Guide*.
- Changes the specification of the type by adding object member subprogram specifications.
- Recompiles the specification or body of the type.
- Resets the version of the type to 1, so that it is no longer considered to be evolved.

Topics

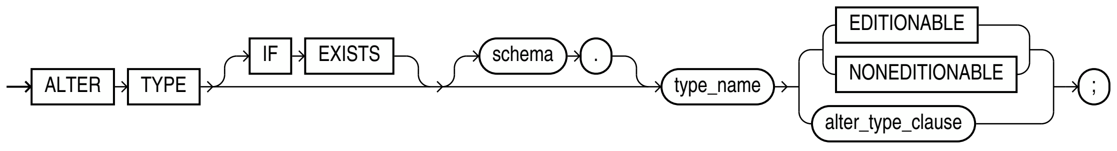
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

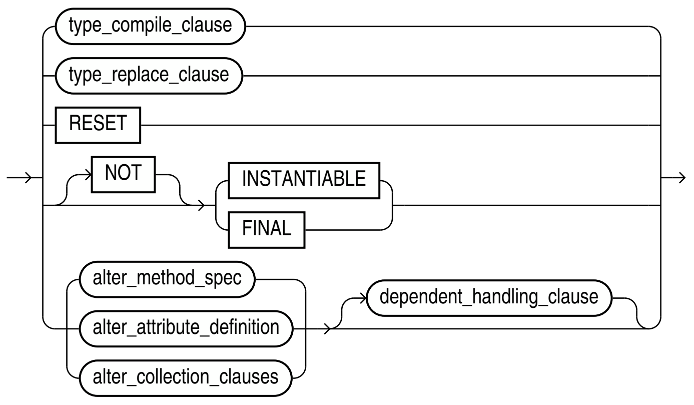
If the type is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the type must be in your schema and you must have `CREATE TYPE` or `CREATE ANY TYPE` system privilege, or you must have `ALTER ANY TYPE` system privileges.

Syntax

alter_type **::=**

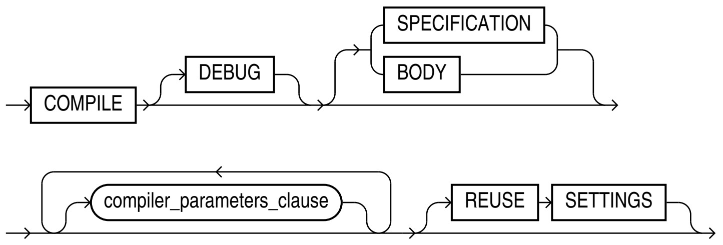


alter_type_clause **::=**



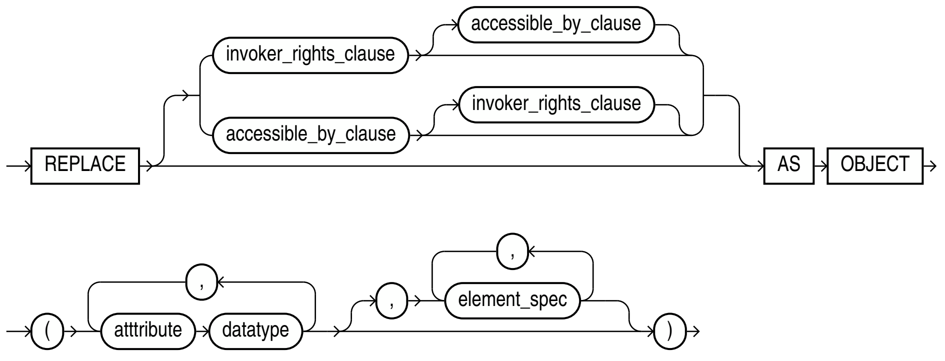
(*type_compile_clause* ::=, *type_replace_clause* ::=, *alter_attribute_definition* ::=, *alter_method_spec* ::=, *alter_collections_clauses*::=, *dependent_handling_clause* ::=)

type_compile_clause **::=**



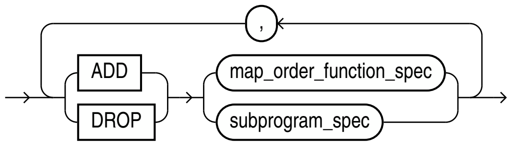
(*compiler_parameters_clause* ::=)

type_replace_clause **::=**



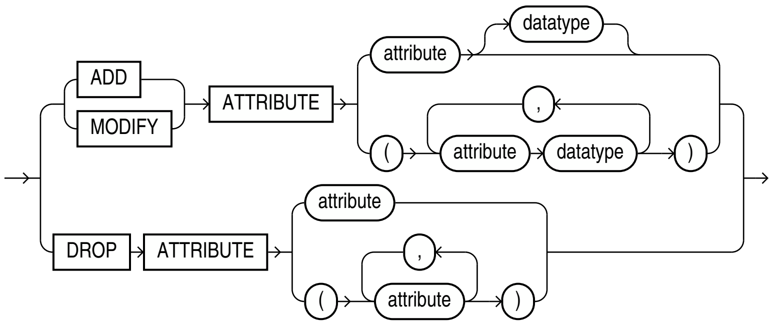
(*accessible_by_clause* ::=, *invoker_rights_clause* ::=, *element_spec* ::=)

alter_method_spec **::=**

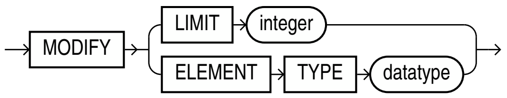


(*map_order_function_spec ::=*, *subprogram_spec ::=*)

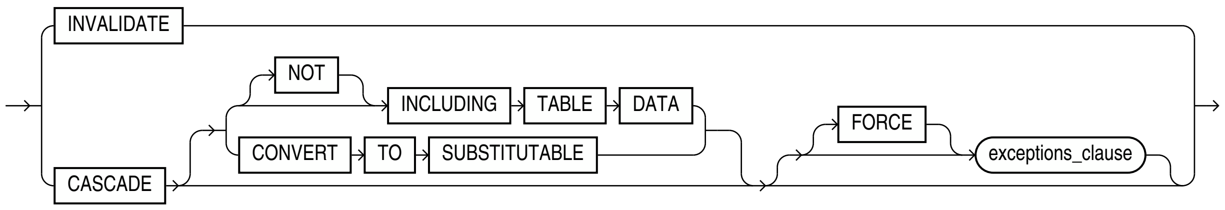
alter_attribute_definition **::=**



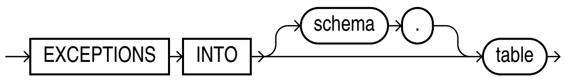
***alter_collections_clauses*::=**



dependent_handling_clause **::=**



exceptions_clause **::=**



Semantics

alter_type

IF EXISTS

Performs the action specified by the *`alter_type_clause`* on the type if it exists. If no such type exists, the statement is ignored without error.

schema

Name of the schema containing the type. **Default:** your schema.

type_name

Name of an ADT, `VARRAY` type, or nested table type.

Restriction on *type_name*

You cannot evolve an editioned ADT.

The `ALTER TYPE` statement fails with ORA-22348 if either of the following is true:

- The type is an editioned ADT and the `ALTER TYPE` statement has no *`type_compile_clause`*.

  (You can use the `ALTER TYPE` statement to recompile an editioned object type, but not for any other purpose.)
- The type has a dependent that is an editioned ADT and the `ALTER TYPE` statement has a `CASCADE` clause.

An **editioned object** is a schema object that has an editionable object type and was created by a user for whom editions are enabled.

{ EDITIONABLE | NONEDITIONABLE }

Specifies whether the type becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `TYPE` in *`schema`*. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

alter_type_clause

RESET

Resets the version of this type to 1, so that it is no longer considered to be evolved.

> **Note:**
>
> Resetting the version of this type to 1 invalidates all of its dependents.

`RESET` is intended for evolved ADTs that are preventing their owners from being editions-enabled. For information about enabling editions for users, see *Oracle Database Development Guide*.

To see the version number of an ADT, select `VERSION#` from the static data dictionary view `*_TYPE_VERSIONS`. For example:

```
SELECT Version#
FROM DBA_TYPE_VERSIONS
```

```
WHERE Owner = schema
AND Name = 'type_name'
AND Type = 'TYPE'
```

For an evolved ADT, the preceding query returns multiple rows with different version numbers. `RESET` deletes every row whose version number is less than the maximum version number, and resets the version number of the remaining rows to 1.

Restriction on RESET

You cannot specify `RESET` if the type has any table dependents (direct or indirect).

[NOT] INSTANTIABLE

Specify `INSTANTIABLE` if object instances of this type can be constructed.

Specify `NOT INSTANTIABLE` if no constructor (default or user-defined) exists for this type. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement).

Restriction on NOT INSTANTIABLE

You cannot change a user-defined type from `INSTANTIABLE` to `NOT INSTANTIABLE` if the type has any table dependents.

[NOT] FINAL

Specify `FINAL` if no further subtypes can be created for this type.

Specify `NOT FINAL` if further subtypes can be created under this type.

If you change the property from `FINAL` to `NOT FINAL`, or the reverse, then you must specify the `CASCADE` clause of the "*dependent_handling_clause*" to convert data in dependent columns and tables. Specifically:

- If you change a type from `NOT FINAL` to `FINAL`, then you must specify `CASCADE` [`INCLUDING TABLE DATA`]. You cannot defer data conversion with `CASCADE NOT INCLUDING TABLE DATA`.
- If you change a type from `FINAL` to `NOT FINAL`, then:
  - Specify `CASCADE INCLUDING TABLE DATA` if you want to create substitutable tables and columns of that type, but you are not concerned about the substitutability of the existing dependent tables and columns.

    The database marks all existing dependent columns and tables `NOT SUBSTITUTABLE AT ALL LEVELS`, so you cannot insert the subtype instances of the altered type into these existing columns and tables.
  - Specify `CASCADE CONVERT TO SUBSTITUTABLE` if you want to create substitutable tables and columns of the type and also store subtype instances of the altered type in existing dependent tables and columns.

    The database marks all existing dependent columns and tables `SUBSTITUTABLE AT ALL LEVELS` except those that are explicitly marked `NOT SUBSTITUTABLE AT ALL LEVELS`.

See Also:

Oracle Database Object-Relational Developer's Guide for a full discussion of ADT evolution

Restriction on FINAL

You cannot change a user-defined type from `NOT FINAL` to `FINAL` if the type has any subtypes.

type_compile_clause

(Default) Recompiles the type specification and body.

See *compile_clause* and *compiler_parameters_clause* semantics.

type_replace_clause

Starting with Oracle Database 12c Release 2 (12.2), the *`type_replace_clause`* is deprecated. Use the *`alter_method_spec`* clause instead. Alternatively, you can recreate the type using the `CREATE OR REPLACE TYPE` statement.
Adds member subprogram specifications.

Restriction on *type_replace_clause*

This clause is valid only for ADTs, not for nested tables or varrays.

attribute

Name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object.

element_spec

Specifies elements of the redefined object.

alter_method_spec

Adds a method to or drops a method from the type. The database disables any function-based indexes that depend on the type.

In one `ALTER TYPE` statement you can add or drop multiple methods, but you can reference each method only once.

ADD

When you add a method, its name must not conflict with any existing attributes in its type hierarchy.

DROP

When you drop a method, the database removes the method from the target type.

Restriction on DROP

You cannot drop from a subtype a method inherited from its supertype. Instead you must drop the method from the supertype.

alter_attribute_definition

Adds, drops, or modifies an attribute of an ADT. In one `ALTER TYPE` statement, you can add, drop, or modify multiple member attributes or methods, but you can reference each attribute or method only once.

ADD ATTRIBUTE

Name of the attribute must not conflict with existing attributes or methods in the type hierarchy. The database adds the attribute to the end of the locally defined attribute list.

If you add the attribute to a supertype, then it is inherited by all of its subtypes. In subtypes, inherited attributes always precede declared attributes. Therefore, you might need to update the mappings of the implicitly altered subtypes after adding an attribute to a supertype.

DROP ATTRIBUTE

When you drop an attribute from a type, the database drops the column corresponding to the dropped attribute and any indexes, statistics, and constraints referencing the dropped attribute.

You need not specify the data type of the attribute you are dropping.

Restrictions on DROP ATTRIBUTE

- You cannot drop an attribute inherited from a supertype. Instead you must drop the attribute from the supertype.
- You cannot drop an attribute that is part of a partitioning, subpartitioning, or cluster key.

> **Caution:**
>
> If you use the `INVALIDATE` option, then the compiler does not check dependents; therefore, this rule is not enforced. However, dropping such an attribute leaves the table in an unusable state.

- You cannot drop an attribute of a primary-key-based object identifier of an object table or a primary key of an index-organized table.
- You cannot drop all of the attributes of a root type. Instead you must drop the type. However, you can drop all of the locally declared attributes of a subtype.

MODIFY ATTRIBUTE

Modifies the data type of an existing scalar attribute. For example, you can increase the length of a `VARCHAR2` or `RAW` attribute, or you can increase the precision or scale of a numeric attribute.

Restriction on MODIFY ATTRIBUTE

You cannot expand the size of an attribute referenced in a function-based index, domain index, or cluster key.

alter_collection_clauses

These clauses are valid only for collection types.

MODIFY LIMIT *integer*

Increases the number of elements in a varray. It is not valid for nested tables. Specify an integer greater than the current maximum number of elements in the varray.

MODIFY ELEMENT TYPE *datatype*

Increases the precision, size, or length of a scalar data type of a varray or nested table. This clause is not valid for collections of ADTs.

- For a collection of `NUMBER`, you can increase the precision or scale.
- For a collection of `RAW`, you can increase the maximum size.
- For a collection of `VARCHAR2` or `NVARCHAR2`, you can increase the maximum length.

dependent_handling_clause

Specifies how the database is to handle objects that are dependent on the modified type. If you omit this clause, then the `ALTER TYPE` statement terminates if the type has any dependent type or table.

INVALIDATE

Invalidates all dependent objects without any checking mechanism. Starting with Oracle Database 12c Release 2 (12.2), the `INVALIDATE` command is deprecated. Oracle recommends that you use the `CASCADE` clause instead.

Caution:

The database does not validate the type change, so use this clause with caution. For example, if you drop an attribute that is a partitioning or cluster key, then the table becomes unusable.

CASCADE

Propagates the type change to dependent types and tables. The database terminates the statement if any errors are found in the dependent types or tables unless you also specify `FORCE`.

If you change the property of the type between `FINAL` and `NOT FINAL`, then you must specify this clause to convert data in dependent columns and tables.

INCLUDING TABLE DATA

(Default) Converts data stored in all user-defined columns to the most recent version of the column type.

Note:

You must specify this clause if your column data is in Oracle database version 8.0 image format. This clause is also required if you are changing the type property between `FINAL` and `NOT FINAL`

- For each attribute added to the column type, the database adds an attribute to the data and initializes it to null.
- For each attribute dropped from the referenced type, the database removes the corresponding attribute data from each row in the table.

If you specify `INCLUDING TABLE DATA`, then all of the tablespaces containing the table data must be in read/write mode.

If you specify `NOT INCLUDING TABLE DATA`, then the database upgrades the metadata of the column to reflect the changes to the type but does not scan the dependent column and update the data as part of this `ALTER TYPE` statement. However, the dependent column data remains accessible, and the results of subsequent queries of the data reflect the type modifications.

CONVERT TO SUBSTITUTABLE

Specify this clause if you are changing the type from `FINAL` to `NOT FINAL` and you want to create substitutable tables and columns of the type and also store subtype instances of the altered type in existing dependent tables and columns.

exceptions_clause

FORCE

Specify `FORCE` if you want the database to ignore the errors from dependent tables and indexes and log all errors in the specified exception table. The exception table must have been created by running the `DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE` procedure.

Examples

See "CREATE TYPE Statement" for examples creating the types referenced in these examples.

Example 15-7 Adding a Member Function

This example uses the ADT `data_typ1`.

A method is added to `data_typ1` and its type body is modified to correspond. The date formats are consistent with the `order_date` column of the `oe.orders` sample table.

```
ALTER TYPE data_typ1
   ADD MEMBER FUNCTION qtr(der_qtr DATE)
   RETURN CHAR CASCADE;

CREATE OR REPLACE TYPE BODY data_typ1 IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
  BEGIN
  RETURN (year + invent);
  END;
     MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
     BEGIN
       IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
         RETURN 'FIRST';
       ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
         RETURN 'SECOND';
       ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
         RETURN 'THIRD';
       ELSE
         RETURN 'FOURTH';
       END IF;
     END;
   END;
/
```

Example 15-8 Adding a Collection Attribute

This example adds the `author` attribute to the `textdoc_tab` object column of the `text` table.

```
CREATE TABLE text (
   doc_id       NUMBER,
   description  textdoc_tab)
   NESTED TABLE description STORE AS text_store;

ALTER TYPE textdoc_typ
   ADD ATTRIBUTE (author VARCHAR2) CASCADE;
```

The `CASCADE` keyword is required because both the `textdoc_tab` and `text` table are dependent on the `textdoc_typ` type.

Example 15-9 Increasing the Number of Elements of a Collection Type

This example increases the maximum number of elements in the varray `phone_list_typ_demo`.

```
ALTER TYPE phone_list_typ_demo
  MODIFY LIMIT 10 CASCADE;
```

Example 15-10 Increasing the Length of a Collection Type

This example increases the length of the varray element type `phone_list_typ`.

```
ALTER TYPE phone_list_typ
  MODIFY ELEMENT TYPE VARCHAR(64) CASCADE;
```

Example 15-11 Recompiling a Type

This example recompiles type `cust_address_typ` in the `hr` schema.

```
ALTER TYPE cust_address_typ2 COMPILE;
```

Example 15-12 Recompiling a Type Specification

This example compiles the type specification of `link2`.

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);
/
CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);
/
CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS
      BEGIN
         dbms_output.put_line(c1);
         RETURN c1;
      END;
   END;
/
```

In this example, both the specification and body of `link2` are invalidated because `link1`, which is an attribute of `link2`, is altered.

```
ALTER TYPE link1 ADD ATTRIBUTE (b NUMBER) INVALIDATE;
```

You must recompile the type by recompiling the specification and body in separate statements:

```
ALTER TYPE link2 COMPILE SPECIFICATION;

ALTER TYPE link2 COMPILE BODY;
```

Alternatively, you can compile both specification and body at the same time:

```
ALTER TYPE link2 COMPILE;
```

Example 15-13 Evolving and Resetting an ADT

This example creates an ADT in the schema `Usr`, evolves that ADT, and then tries to enable editions for `Usr`, which fails.

Then the example resets the version of the ADT to 1 and succeeds in enabling editions for `Usr`. To show the version numbers of the newly created, evolved, and reset ADT, the example uses the static data dictionary view `DBA_TYPE_VERSIONS`.

```
-- Create ADT in schema Usr:
create type Usr.My_ADT authid Definer is object(a1 number)

-- Show version number of ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

Result:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)


1 row selected.

-- Evolve ADT:
alter type Usr.My_ADT add attribute (a2 number)
/

-- Show version number of evolved ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

Result:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)

2
type     My_ADT authid Definer is object(a1 number)

2
 alter type     My_ADT add attribute (a2 number)


3 rows selected.

-- Try to enable editions for Usr:
alter user Usr enable editions
/
```

Result:

```
alter user Usr enable editions
*
ERROR at line 1:
ORA-38820: user has evolved object type

-- Reset version of ADT to 1:
```

```
alter type Usr.My_ADT reset
/

-- Show version number of reset ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

Result:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)

1
 alter type     My_ADT add attribute (a2 number)


2 rows selected.

-- Try to enable editions for Usr:
alter user Usr enable editions
/
```

Result:

```
User altered.
```

Related Topics

In this chapter:

- "CREATE TYPE Statement"
- "CREATE TYPE BODY Statement"
- "DROP TYPE Statement"

In other books:

- *Oracle Database Development Guide* for more information about editions
- *Oracle Database Development Guide* for more information about pragmas
- *Oracle Database Object-Relational Developer's Guide* for more information about the implications of not including table data when modifying type attribute

CREATE FUNCTION Statement

The `CREATE FUNCTION` statement creates or replaces a standalone function or a call specification.

A **standalone function** is a function (a subprogram that returns a single value) that is stored in the database.

Note:

A standalone function that you create with the `CREATE FUNCTION` statement differs from a function that you declare and define in a PL/SQL block or package. For more information, see "Function Declaration and Definition" and CREATE PACKAGE Statement.

A **call specification** declares a Java method, a C function, or a JavaScript function 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 function 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.

Note:

To be callable from SQL statements, a stored function must obey certain rules that control side effects. See "Subprogram Side Effects".

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

To create or replace a standalone function in your schema, you must have the `CREATE PROCEDURE` system privilege.

To create or replace a standalone function in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

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

To embed a `CREATE FUNCTION` statement inside an Oracle precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

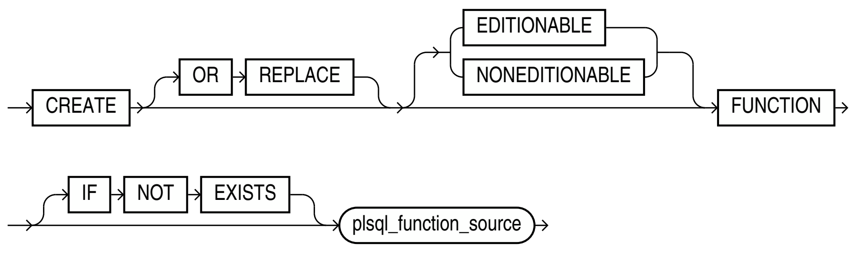
See Also:

For more information about such prerequisites:

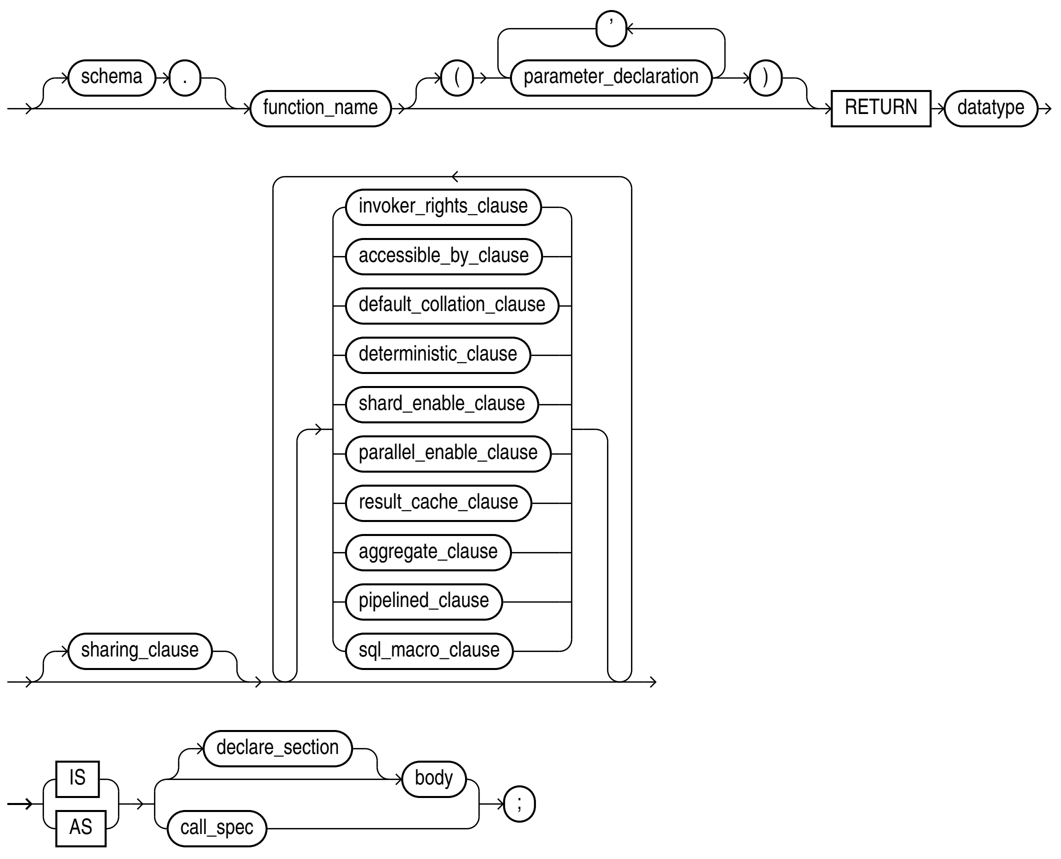
- *Oracle Database Development Guide*
- *Oracle Database Java Developer's Guide*
- *Oracle Database JavaScript Developer's Guide*

Syntax

create_function **::=**



plsql_function_source **::=**



(*sharing_clause* ::= , *invoker_rights_clause* ::= , *accessible_by_clause* ::= , *default_collation_clause* ::= , *deterministic_clause* ::= , *shard_enable_clause* ::= , *parallel_enable_clause* ::= , *result_cache_clause* ::= , *aggregate_clause* ::= , *pipelined_clause* ::=, *sql_macro_clause* ::= , *body* ::= , *call_spec* ::= , *datatype* ::= , *declare_section* ::= , *parameter_declaration* ::=)

Semantics

create_function

OR REPLACE

Re-creates the function if it exists, and recompiles it.

Users who were granted privileges on the function before it was redefined can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, then the database marks the indexes `DISABLED`.

[EDITIONABLE | NONEDITIONABLE]

Specifies whether the function is an editioned or noneditioned object if editioning is enabled for the schema object type `FUNCTION` in *`schema`*. **Default:** `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

IF NOT EXISTS

Creates the function if it does not already exist. If a function by the same name does exist, the statement is ignored without error and the original function body remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_function_source

schema

Name of the schema containing the function. **Default:** your schema.

function_name

Name of the function to be created.

Note:

If you plan to invoke a stored subprogram using a stub generated by SQL*Module, then the stored subprogram name must also be a legal identifier in the invoking host 3GL language, such as Ada or C.

RETURN *datatype*

For *`datatype`*, specify the data type of the return value of the function. The return value can have any data type supported by PL/SQL.

The data type cannot specify a length, precision, or scale. The database derives the length, precision, or scale of the return value from the environment from which the function is called.

If the return type is `ANYDATASET` and you intend to use the function in the `FROM` clause of a query, then you must also specify the `PIPELINED` clause and define a describe method (`ODCITableDescribe`) as part of the implementation type of the function.

You cannot constrain this data type (with `NOT NULL`, for example).

body

The required executable part of the function and, optionally, the exception-handling part of the function.

declare_section

The optional declarative part of the function. Declarations are local to the function, can be referenced in *body*, and cease to exist when the function completes execution.

call_spec

The reference to a call specification mapping a C procedure, Java method name, or JavaScript function name, parameter types, and return type to their SQL counterparts.

Examples

Example 15-14 Creating a Function

This statement creates the function `get_bal` on the sample table `oe.orders`.

```
CREATE FUNCTION IF NOT EXISTS get_bal(acc_no IN NUMBER)
   RETURN NUMBER
   IS acc_bal NUMBER(11,2);
   BEGIN
      SELECT order_total
      INTO acc_bal
      FROM orders
      WHERE customer_id = acc_no;
      RETURN(acc_bal);
    END;
/
```

The `get_bal` function returns the balance of a specified account.

When you invoke the function, you must specify the argument `acc_no`, the number of the account whose balance is sought. The data type of `acc_no` is `NUMBER`.

The function returns the account balance. The `RETURN` clause of the `CREATE FUNCTION` statement specifies the data type of the return value to be `NUMBER`.

The function uses a `SELECT` statement to select the `balance` column from the row identified by the argument `acc_no` in the `orders` table. The function uses a `RETURN` statement to return this value to the environment in which the function is called.

The optional `IF NOT EXISTS` clause is used to ensure that the statement is idempotent. The resulting output message (in this case `Function created.`) is the same whether the function is created or the statement is ignored.

The function created in the preceding example can be used in a SQL statement. For example:

```
SELECT get_bal(165) FROM DUAL;

GET_BAL(165)
------------
        2519
```

Example 15-15   Creating Aggregate Functions

The next statement creates an aggregate function called `SecondMax` to aggregate over number values. It assumes that the ADT `SecondMaxImpl` subprograms contains the implementations of the `ODCIAggregate` subprograms:

```
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER
    PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

See Also:

Oracle Database Data Cartridge Developer's Guide for the complete implementation of type and type body for `SecondMaxImpl`

Use such an aggregate function in a query like this statement, which queries the sample table `hr.employees`:

```
SELECT SecondMax(salary) "SecondMax", department_id
      FROM employees
      GROUP BY department_id
      HAVING SecondMax(salary) > 9000
      ORDER BY "SecondMax", department_id;

SecondMax  DEPARTMENT_ID
---------- -------------
     13500            80
     17000            90
```

Example 15-16   Package Procedure in a Function

This statement creates a function that uses a `DBMS_LOB.GETLENGTH` procedure to return the length of a `CLOB` column.

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
   RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

Example 15-17   Creating Functions Using MLE Module and Inline Call Specifications

In this example, the same function is created in JavaScript twice. Once using an inline call specification and the other using an MLE module.

The following statement creates a JavaScript function with its declaration inline:

```
CREATE OR REPLACE FUNCTION hello_inline(
  "who" VARCHAR2
) RETURN VARCHAR2
AS MLE LANGUAGE JAVASCRIPT
{{
  return `Hello, ${who}`;
}};
/
```

You can then call the function, for example, in an anonymous block:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(hello_inline('Jane'));
END;
/

Hello, Jane
```

The following statements first create an MLE module that implements the `hello` function and then publish the function using a call specification:

```
CREATE OR REPLACE MLE MODULE hello_mod
LANGUAGE JAVASCRIPT AS
  export function hello(who){
    return `Hello, ${who}`;
  }
/

CREATE OR REPLACE FUNCTION hello(
  "p_who" VARCHAR2
) RETURN VARCHAR2
AS MLE MODULE hello_mod
SIGNATURE 'hello';
/
```

The following is the result of a call to the `hello` function:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(hello('Emma'));
END;
/

Hello, Emma
```

Related Topics

In this chapter:

- "ALTER FUNCTION Statement"
- "CREATE PROCEDURE Statement"
- "DROP FUNCTION Statement"

In other chapters:

- Overview of Polymorphic Table Functions
- "Function Declaration and Definition" for information about creating a function in a PL/SQL block
- "Formal Parameter Declaration"
- "PL/SQL 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 Development Guide* for more information about call specifications
- *Oracle Database Data Cartridge Developer's Guide* for information about defining the `ODCITableDescribe` function
- *Oracle Database JavaScript Developer's Guide* for information about call specifications for MLE modules and inline MLE call specifications
- *Oracle Database Java Developer's Guide* for information about call specifications for Java stored procedures

CREATE LIBRARY Statement

The `CREATE LIBRARY` statement creates a **library**, which is a schema object associated with an operating-system shared library.

Note:

The `CREATE LIBRARY` statement is valid only on platforms that support shared libraries and dynamic linking.

For instructions for creating an operating-system shared library, or DLL, see *Oracle Database Development Guide*.

You can use the name of the library schema object in the *`call_spec`* of `CREATE FUNCTION` or `CREATE PROCEDURE` statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can invoke third-generation-language (3GL) functions and procedures.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

To create a library in your schema, you must have the `CREATE LIBRARY` system privilege. To create a library in another user's schema, you must have the `CREATE ANY LIBRARY` system privilege.

To create a library that is associated with a DLL in a directory object, you must have the `EXECUTE` object privilege on the directory object.

To create a library that is associated with a credential name, you must have the `EXECUTE` object privilege on the credential name.

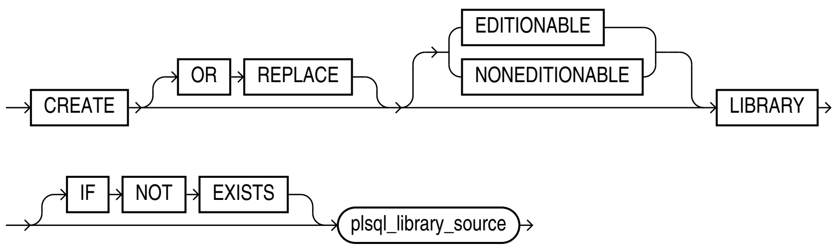
To use the library in the `call_spec` of a `CREATE FUNCTION` statement, or when declaring a function in a package or type, you must have the `EXECUTE` object privilege on the library and the `CREATE FUNCTION` system privilege.

To use the library in the `call_spec` of a `CREATE PROCEDURE` statement, or when declaring a procedure in a package or type, you must have the `EXECUTE` object privilege on the library and the `CREATE PROCEDURE` system privilege.

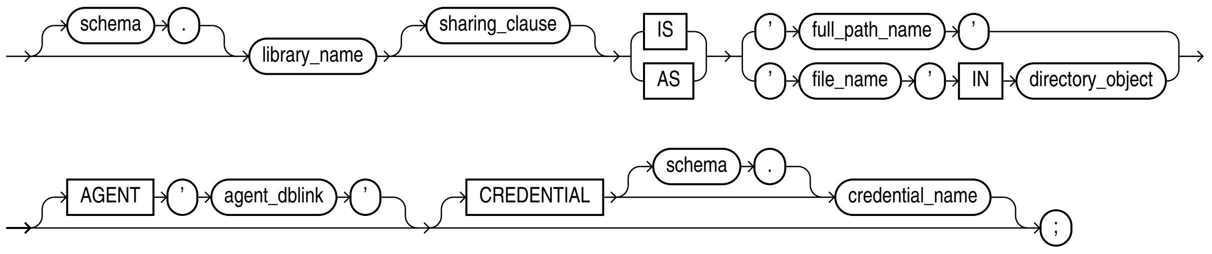
To run a procedure or function defined with the `call_spec` (including a procedure or function defined within a package or type), you must have the `EXECUTE` object privilege on the procedure or function (but you do not need the `EXECUTE` object privilege on the library).

Syntax

***create_library* ::=**



***plsql_library_source* ::=**



(*sharing_clause* ::=)

Semantics

create_library

OR REPLACE

Re-creates the library if it exists, and recompiles it.

Users who were granted privileges on the library before it was redefined can still access it without being regranted the privileges.

[EDITIONABLE | NONEDITIONABLE]

Specifies whether the library is an editioned or noneditioned object if editioning is enabled for the schema object type `LIBRARY` in *`schema`*. **Default:** `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

IF NOT EXISTS

Creates the library if it does not already exist. If a library by the same name does exist, the statement is ignored without error and the original library remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_library_source

schema

Name of the schema containing the library. **Default:** your schema.

library_name

Name that represents this library when a user declares a function or procedure with a *`call_spec`*.

'*full_path_name*'

String literal enclosed in single quotation marks, whose value your operating system recognizes as the full path name of a shared library.

The *`full_path_name`* is not interpreted during execution of the `CREATE LIBRARY` statement. The existence of the shared library is checked when someone invokes one of its subprograms.

'*file_name*' IN *directory_object*

The *`file_name`* is a string literal enclosed in single quotation marks, whose value is the name of a dynamic link library (DLL) in *`directory_object`*. The string literal cannot exceed 2,000 bytes and cannot contain path delimiters. The compiler ignores *`file_name`*, but at run time, *`file_name`* is checked for path delimiters.

`directory_object`

The *`directory_object`* is a directory object, created with the `CREATE DIRECTORY` statement (described in *Oracle Database SQL Language Reference*). If *`directory_object`* does not exist or you do not have the `EXECUTE` object privilege on *`directory_object`*, then the library is created with errors. If *`directory_object`* is subsequently created, then the library becomes invalid. Other reasons that the library can become invalid are:

- *`directory_object`* is dropped.
- *`directory_object`* becomes invalid.
- Your `EXECUTE` object privilege on *`directory_object`* is revoked.

If you create a library object in a PDB that has a predefined `PATH_PREFIX`, the library must use a directory object. The directory object will enforce the rules of `PATH_PREFIX` for the library object. Failure to use a directory object in the library object will raise a compilation error.

If a database is plugged into a CDB as a PDB with a predefined `PATH_PREFIX`, attempts to use a library object that does not use a directory object result in an ORA-65394 error. The library object will not be invalidated, but to make it usable, you must recreate it using a directory object. See *Oracle Multitenant Administrator's Guide* for more information about CDB administration.

AGENT '*agent_dblink*'

Causes external procedures to run from a database link other than the server. Oracle Database uses the database link that *`agent_dblink`* specifies to run external procedures. If you omit this clause, then the default agent on the server (`extproc`) runs external procedures.

CREDENTIAL [*schema*.]*credential_name*

Specifies the credentials of the operating system user that the `extproc` agent impersonates when running an external subprogram that specifies the library. **Default:** Owner of the Oracle Database installation.

If *`credential_name`* does not exist or you do not have the `EXECUTE` object privilege on *`credential_name`*, then the library is created with errors. If *`credential_name`* is subsequently created, then the library becomes invalid. Other reasons that the library can become invalid are:

- *`credential_name`* is dropped.
- *`credential_name`* becomes invalid.
- Your `EXECUTE` object privilege on *`credential_name`* is revoked.

For information about using credentials, see *Oracle Database Security Guide*.

Examples

Example 15-18 Creating a Library

The following statement creates library `ext_lib`, using a directory object:

```
CREATE LIBRARY IF NOT EXISTS ext_lib AS 'ddl_1' IN ddl_dir;
/
```

The optional `IF NOT EXISTS` clause is used to ensure that the statement is idempotent. The resulting output message (in this case `Library created`) is the same whether the library is created or the statement is ignored.

The following statement re-creates library `ext_lib`, using a directory object and a credential:

```
CREATE OR REPLACE LIBRARY ext_lib AS 'ddl_1' IN ddl_dir CREDENTIAL ddl_cred;
/
```

The following statement creates library `ext_lib`, using an explicit path:

```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
/
```

The following statement re-creates library `ext_lib`, using an explicit path:

```
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
/
```

Example 15-19 Specifying an External Procedure Agent

The following example creates a library `app_lib` (using an explicit path) and specifies that external procedures run from the public database `sales.hq.example.com`:

```
CREATE LIBRARY app_lib as '${ORACLE_HOME}/lib/app_lib.so'
   AGENT 'sales.hq.example.com';
/
```

See Also:

Oracle Database SQL Language Reference for information about creating database links

Related Topics

- "ALTER LIBRARY Statement"
- "DROP LIBRARY Statement"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"

CREATE PACKAGE Statement

The `CREATE PACKAGE` statement creates or replaces the specification for a stored **package**, which is an encapsulated collection of related procedures, functions, and other program objects stored as a unit in the database.

The **package specification** declares these objects. The **package body**, specified subsequently, defines these objects.

Topics

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

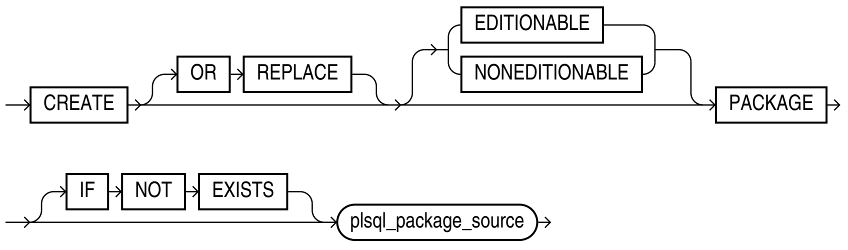
Prerequisites

To create or replace a package in your schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

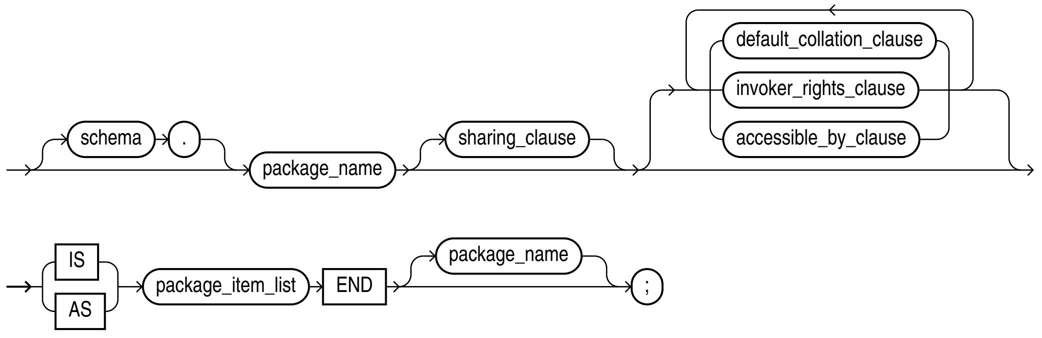
To embed a `CREATE PACKAGE` statement inside an Oracle database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

Syntax

create_package **::=**

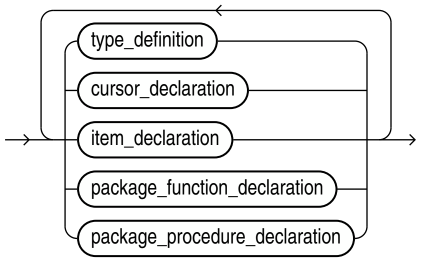


plsql_package_source **::=**



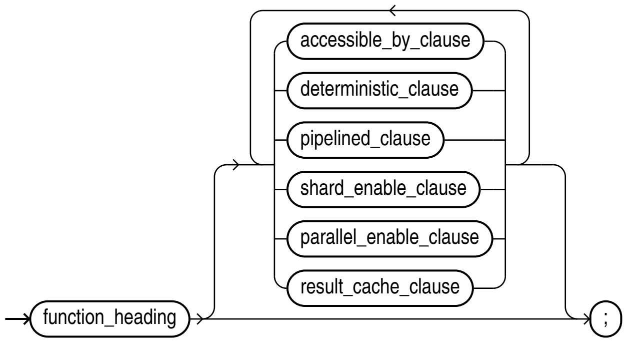
(*sharing_clause* ::= , *default_collation_clause* ::= , *invoker_rights_clause* ::= , *accessible_by_clause* ::= ,)

package_item_list **::=**



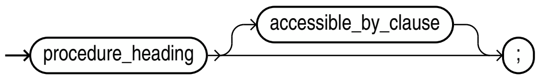
(*cursor_declaration* ::= , *item_declaration* ::= , *type_definition* ::=)

package_function_declaration **::=**



(*function_heading* ::= , *accessible_by_clause* ::= , *deterministic_clause* ::= , *pipelined_clause* ::= , *shard_enable_clause* ::= , *parallel_enable_clause* ::= , *result_cache_clause* ::=)

***package_procedure_declaration* ::=**



(*procedure_heading* ::=, *accessible_by_clause* ::=)

Semantics

create_package

OR REPLACE

Re-creates the package if it exists, and recompiles it.

Users who were granted privileges on the package before it was redefined can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, then the database marks the indexes `DISABLED`.

[EDITIONABLE | NONEDITIONABLE]

Specifies whether the package is an editioned or noneditioned object if editioning is enabled for the schema object type `PACKAGE` in *`schema`*. **Default:** `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

IF NOT EXISTS

Creates the package if it does not already exist. If a package by the same name does exist, the statement is ignored without error and the original package remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_package_source

schema

Name of the schema containing the package. **Default:** your schema.

package_name

A package stored in the database. For naming conventions, see "Identifiers".

package_item_list

Defines every type in the package and declares every cursor and subprogram in the package. Except for polymorphic table functions, every declaration must have a corresponding definition in the package body. The headings of corresponding declarations and definitions must match word for word, except for white space. Package polymorphic table function must be declared in the same package as their implementation package.

Restriction on *package_item_list*

`PRAGMA AUTONOMOUS_TRANSACTION` cannot appear here.

Example

Example 15-20 Creating the Specification for the emp_mgmt Package

This statement creates the specification of the `emp_mgmt` package.

```
CREATE PACKAGE IF NOT EXISTS emp_mgmt AS
   FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2,
      manager_id NUMBER, salary NUMBER,
      commission_pct NUMBER, department_id NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
      RETURN NUMBER;
   PROCEDURE remove_emp(employee_id NUMBER);
   PROCEDURE remove_dept(department_id NUMBER);
   PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER);
   PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER);
   no_comm EXCEPTION;
   no_sal EXCEPTION;
END emp_mgmt;
/
```

The specification for the `emp_mgmt` package declares these public program objects:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`
- The exceptions `no_comm` and `no_sal`

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that invoke any of these public procedures or functions or raise any of the public exceptions of the package.

The optional `IF NOT EXISTS` clause is used to ensure that the statement is idempotent. The resulting output message (in this case `Package created`) is the same whether the package is created or the statement is ignored.

Before you can invoke this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a `CREATE PACKAGE BODY` statement that creates the body of the `emp_mgmt` package, see "CREATE PACKAGE BODY Statement".

Related Topics

In this chapter:

- "ALTER PACKAGE Statement"
- "CREATE PACKAGE Statement"
- "CREATE PACKAGE BODY Statement"
- "DROP PACKAGE Statement"

In other chapters:

- "PL/SQL Packages"
- "Package Specification"

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

CREATE PACKAGE BODY Statement

The `CREATE PACKAGE BODY` statement creates or replaces the body of a stored **package**, which is an encapsulated collection of related procedures, stored functions, and other program objects stored as a unit in the database.

The **package body** defines these objects. The **package specification**, defined in an earlier `CREATE PACKAGE` statement, declares these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

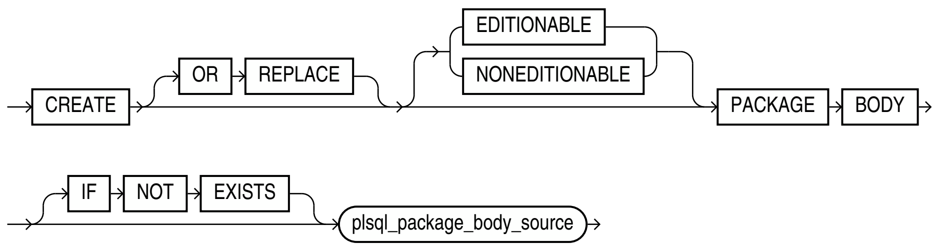
Prerequisites

To create or replace a package in your schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege. In both cases, the package body must be created in the same schema as the package.

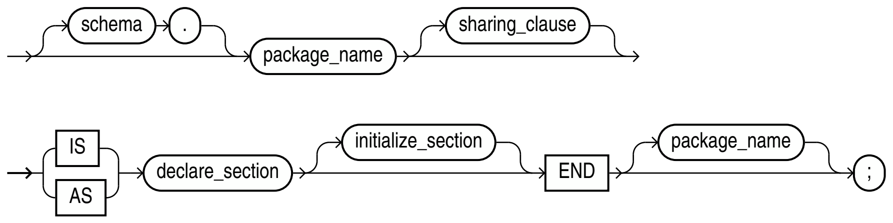
To embed a `CREATE PACKAGE BODY` statement inside an the database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

Syntax

create_package_body **::=**

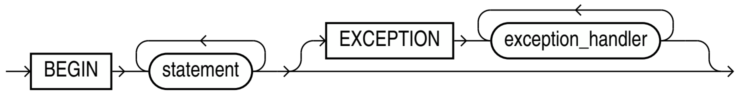


***plsql_package_body_source* ::=**



(*sharing_clause* ::=, *declare_section* ::=)

***initialize_section* ::=**



(*statement* ::= , *exception_handler* ::=)

Semantics

create_package_body

OR REPLACE

Re-creates the package body if it exists, and recompiles it.

Users who were granted privileges on the package body before it was redefined can still access the package without being regranted the privileges.

[EDITIONABLE | NONEDITIONABLE]

If you do not specify this property, then the package body inherits `EDITIONABLE` or `NONEDITIONABLE` from the package specification. If you do specify this property, then it must match that of the package specification.

IF NOT EXISTS

Creates the package body if it does not already exist. If a package body by the same name does exist, the statement is ignored without error and the original package body remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_package_body_source

schema

Name of the schema containing the package. **Default:** your schema.

package_name

Name of the package to be created.

declare_section

Has a definition for every cursor and subprogram declaration in the package specification. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

Can also declare and define private items that can be referenced only from inside the package.

Restriction on *declare_section*

`PRAGMA AUTONOMOUS_TRANSACTION` cannot appear here.

initialize_section

Initializes variables and does any other one-time setup steps.

Examples

Example 15-21 Creating the emp_mgmt Package Body

This statement creates the body of the `emp_mgmt` package created in "Example 15-20".

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
   tot_emps NUMBER;
   tot_depts NUMBER;
FUNCTION hire
   (last_name VARCHAR2, job_id VARCHAR2,
    manager_id NUMBER, salary NUMBER,
    commission_pct NUMBER, department_id NUMBER)
   RETURN NUMBER IS new_empno NUMBER;
BEGIN
   SELECT employees_seq.NEXTVAL
      INTO new_empno
      FROM DUAL;
   INSERT INTO employees
      VALUES (new_empno, 'First', 'Last','first.example@example.com',
              '(415)555-0100',
              TO_DATE('18-JUN-2002','DD-MON-YYYY'),
              'IT_PROG',90000000,00, 100,110);
      tot_emps := tot_emps + 1;
   RETURN(new_empno);
END;
FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
   RETURN NUMBER IS
      new_deptno NUMBER;
   BEGIN
      SELECT departments_seq.NEXTVAL
         INTO new_deptno
         FROM dual;
      INSERT INTO departments
         VALUES (new_deptno, 'department name', 100, 1700);
      tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;
PROCEDURE remove_emp (employee_id NUMBER) IS
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
      tot_emps := tot_emps - 1;
   END;
PROCEDURE remove_dept(department_id NUMBER) IS
   BEGIN
      DELETE FROM departments
      WHERE departments.department_id = remove_dept.department_id;
      tot_depts := tot_depts - 1;
      SELECT COUNT(*) INTO tot_emps FROM employees;
   END;
PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER) IS
   curr_sal NUMBER;
   BEGIN
      SELECT salary INTO curr_sal FROM employees
      WHERE employees.employee_id = increase_sal.employee_id;
      IF curr_sal IS NULL
         THEN RAISE no_sal;
      ELSE
         UPDATE employees
         SET salary = salary + salary_incr
         WHERE employee_id = employee_id;
      END IF;
   END;
PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER) IS
   curr_comm NUMBER;
   BEGIN
      SELECT commission_pct
      INTO curr_comm
      FROM employees
      WHERE employees.employee_id = increase_comm.employee_id;
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE employees
         SET commission_pct = commission_pct + comm_incr;
      END IF;
   END;
END emp_mgmt;
```

The package body defines the public program objects declared in the package specification:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`

These objects are declared in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure `increase_all_comms` separate from the `emp_mgmt` package that invokes the `increase_comm` procedure.

These objects are defined in the package body, so you can change their definitions without causing the database to invalidate dependent schema objects. For example, if you subsequently change the definition of `hire`, then the database need not recompile `increase_all_comms` before running it.

The package body in this example also declares private program objects, the variables `tot_emps` and `tot_depts`. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable `tot_depts`. However, the function `create_dept` is part of the package, so `create_dept` can change the value of `tot_depts`.

Related Topics

In this chapter:

- "CREATE PACKAGE Statement"

In other chapters:

- "PL/SQL Packages"
- "Package Body"
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"

CREATE PROCEDURE Statement

The `CREATE PROCEDURE` statement creates or replaces a standalone procedure or a call specification.

A **standalone procedure** is a procedure (a subprogram that performs a specific action) that is stored in the database.

Note:

A standalone procedure that you create with the `CREATE PROCEDURE` statement differs from a procedure that you declare and define in a PL/SQL block or package. For information, see "Procedure Declaration and Definition" or "CREATE PACKAGE Statement".

A **call specification** declares a Java method, a C function, or a JavaScript function so that it can be called 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 procedure 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.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

To create or replace a standalone procedure in your schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a standalone procedure in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

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

To embed a `CREATE PROCEDURE` statement inside an Oracle precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

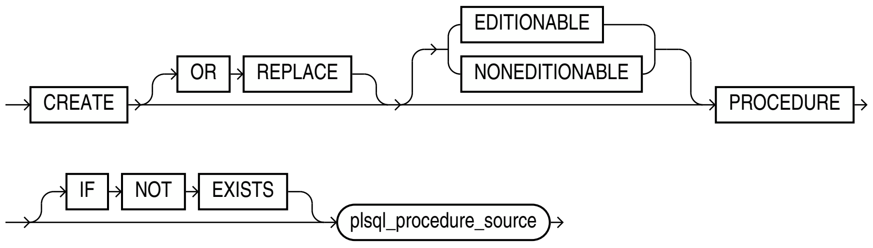
See Also:

For more information about such prerequisites:

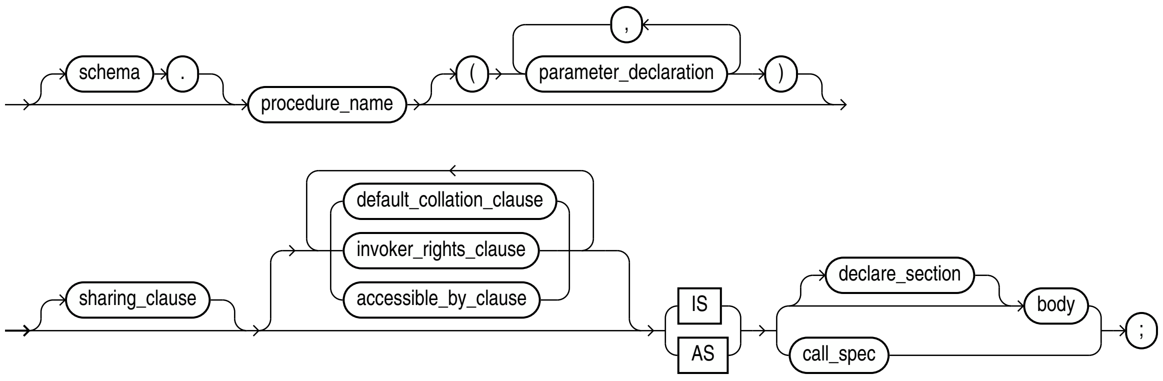
- *Oracle Database Development Guide*
- *Oracle Database Java Developer's Guide*
- *Oracle Database JavaScript Developer's Guide*

Syntax

create_procedure **::=**



plsql_procedure_source **::=**



(*sharing_clause* ::=, *default_collation_clause* ::=, *invoker_rights_clause* ::=, *accessible_by_clause* ::=, *call_spec* ::=, *body* ::=, *declare_section* ::=, *parameter_declaration* ::=)

Semantics

create_procedure

OR REPLACE

Re-creates the procedure if it exists, and recompiles it.

Users who were granted privileges on the procedure before it was redefined can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the procedure, then the database marks the indexes `DISABLED`.

[EDITIONABLE | NONEDITIONABLE]

Specifies whether the procedure is an editioned or noneditioned object if editioning is enabled for the schema object type `PROCEDURE` in *`schema`*. **Default:** `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

IF NOT EXISTS

Creates the procedure if it does not already exist. If a procedure by the same name does exist, the statement is ignored without error and the original procedure remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_procedure_source

schema

Name of the schema containing the procedure. **Default:** your schema.

procedure_name

Name of the procedure to be created.

Note:

If you plan to invoke a stored subprogram using a stub generated by SQL*Module, then the stored subprogram name must also be a legal identifier in the invoking host 3GL language, such as Ada or C.

body

The required executable part of the procedure and, optionally, the exception-handling part of the procedure.

declare_section

The optional declarative part of the procedure. Declarations are local to the procedure, can be referenced in *`body`*, and cease to exist when the procedure completes execution.

call_spec

The reference to a call specification mapping a C procedure, Java method name, or JavaScript function name, parameter types, and return type to their SQL counterparts.

Examples

Example 15-22 Creating a Procedure

This statement creates the procedure `remove_emp` in the schema `hr`.

```
CREATE PROCEDURE IF NOT EXISTS remove_emp (employee_id NUMBER) AS
   tot_emps NUMBER;
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
   tot_emps := tot_emps - 1;
   END;
/
```

The `remove_emp` procedure removes a specified employee. When you invoke the procedure, you must specify the `employee_id` of the employee to be removed.

The procedure uses a `DELETE` statement to remove from the `employee`s table the row of `employee_id`.

The optional `IF NOT EXISTS` clause is used to ensure that the statement is idempotent. The resulting output message (in this case `Procedure created`) is the same whether the procedure is created or the statement is ignored.

See Also:

"CREATE PACKAGE BODY Statement" to see how to incorporate this procedure into a package

Example 15-23 Creating an External Procedure

In this example, external procedure `c_find_root` expects a pointer as a parameter. Procedure `find_root` passes the parameter by reference using the `BY REFERENCE` phrase.

```
CREATE PROCEDURE find_root
   ( x IN REAL )
   IS LANGUAGE C
      NAME c_find_root
      LIBRARY c_utils
      PARAMETERS ( x BY REFERENCE );
```

Example 15-24 Creating Procedures Using MLE Module and Inline Call Specifications

In this example, the same procedure is created in JavaScript twice. Once using an inline call specification and the other using an MLE module.

The following statement creates a JavaScript function with its declaration inline:

```
CREATE OR REPLACE PROCEDURE hello_inline(
  "who" VARCHAR2
)
AS MLE LANGUAGE JAVASCRIPT
{{
  console.log(`Hello, ${who}`);
}};
/
```

You can then call the procedure, as in the following:

```
EXEC hello_inline('Angela');
```

Result:

```
Hello, Angela
```

The following statements first create an MLE module that implements the `hello` function and then publish the procedure using a call specification:

```
CREATE OR REPLACE MLE MODULE hello_mod
LANGUAGE JAVASCRIPT AS
  export function hello(who){
    console.log(`Hello, ${who}`);
  }
/

CREATE OR REPLACE PROCEDURE hello(
  "p_who" VARCHAR2
)
AS MLE MODULE hello_mod
SIGNATURE 'hello';
/
```

The following is an example of a call to the `hello` procedure:

```
EXEC hello('Chris');
```

Result:

```
Hello, Chris
```

Related Topics

In this chapter:

- "ALTER PROCEDURE Statement"
- "CREATE FUNCTION Statement"
- "DROP PROCEDURE Statement"

In other chapters:

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

In other books:

- *Oracle Database SQL Language Reference* for information about the `CALL` statement
- *Oracle Database Development Guide* for more information about call specifications
- *Oracle Database Development Guide* for more information about invoking stored PL/SQL subprograms
- *Oracle Database JavaScript Developer's Guide* for information about call specifications for MLE modules and inline MLE call specifications

- *Oracle Database Java Developer's Guide* for information about call specifications for Java stored procedures

CREATE TRIGGER Statement

The `CREATE TRIGGER` statement creates or replaces a **database trigger**, which is either of these:

- A stored PL/SQL block associated with a table, a view, a schema, or the database
- An anonymous PL/SQL block or an invocation of a procedure implemented in PL/SQL or Java

The database automatically runs a trigger when specified conditions occur.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

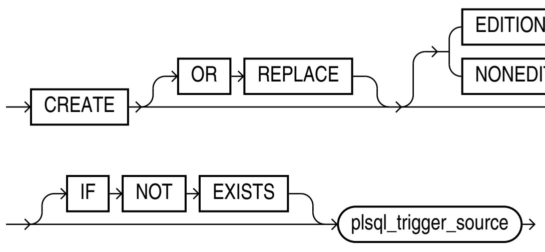
Prerequisites

- To create a trigger in your schema on a table in your schema or on your schema (`SCHEMA`), you must have the `CREATE TRIGGER` system privilege.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*`schema`*`.SCHEMA`), you must have the `CREATE ANY TRIGGER` system privilege.
- In addition to the preceding privileges, to create a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege.
- To create a trigger on a pluggable database (PDB), you must be connected to that PDB and have the `ADMINISTER DATABASE TRIGGER` system privilege. For information about PDBs, see *Oracle Database Administrator's Guide*.
- In addition to the preceding privileges, to create a crossedition trigger, you must be enabled for editions. For information about enabling editions for a user, see *Oracle Database Development Guide*.

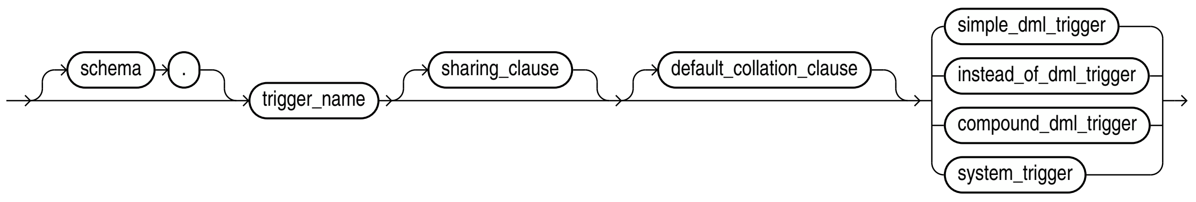
If the trigger issues SQL statements or invokes procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.

Syntax

create_trigger **::=**

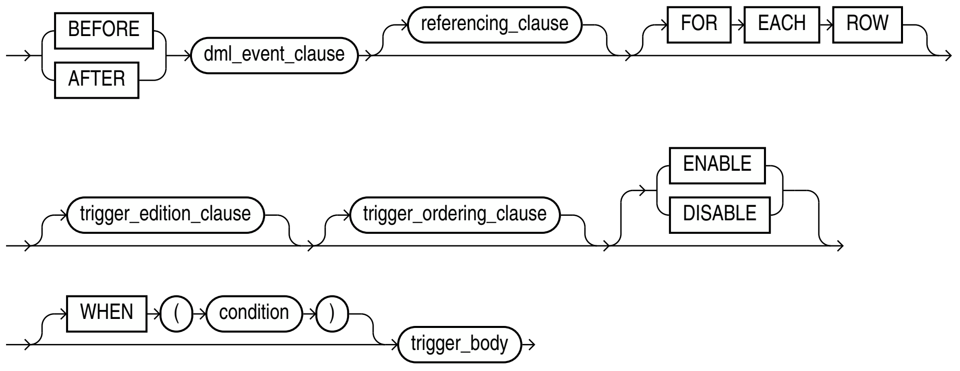


plsql_trigger_source **::=**



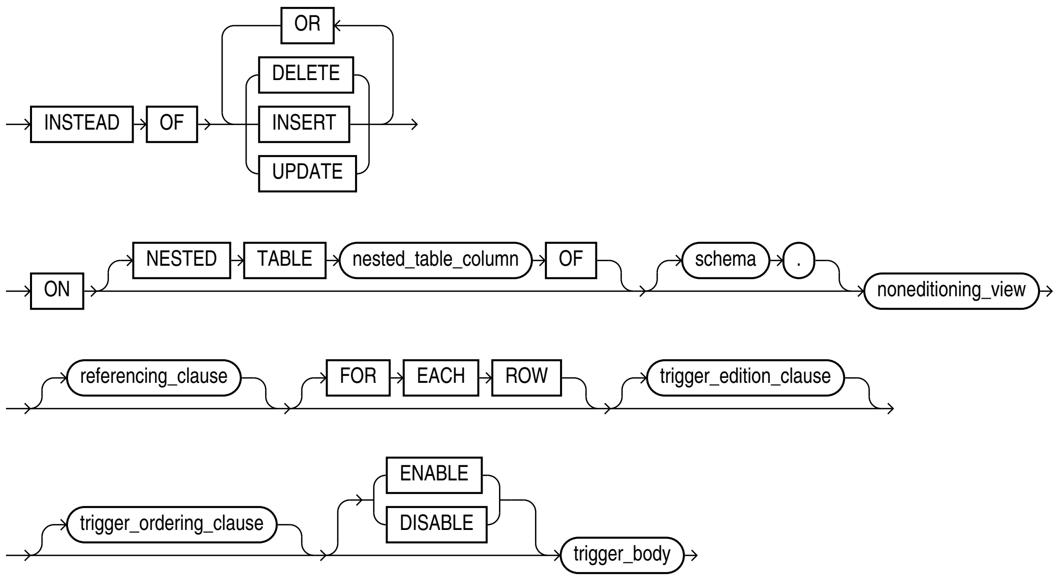
(*sharing_clause ::=* , *default_collation_clause ::=* , *compound_dml_trigger ::=* , *instead_of_dml_trigger ::=* , *system_trigger ::=*)

simple_dml_trigger **::=**



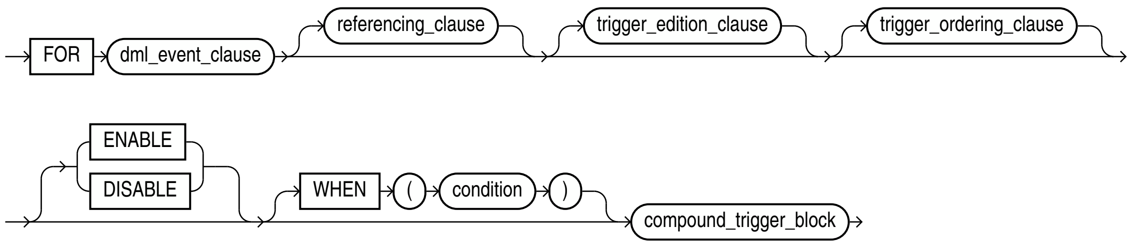
(*dml_event_clause ::=* , *referencing_clause ::=* , *trigger_body ::=* , *trigger_edition_clause ::=* , *trigger_ordering_clause ::=*)

instead_of_dml_trigger **::=**



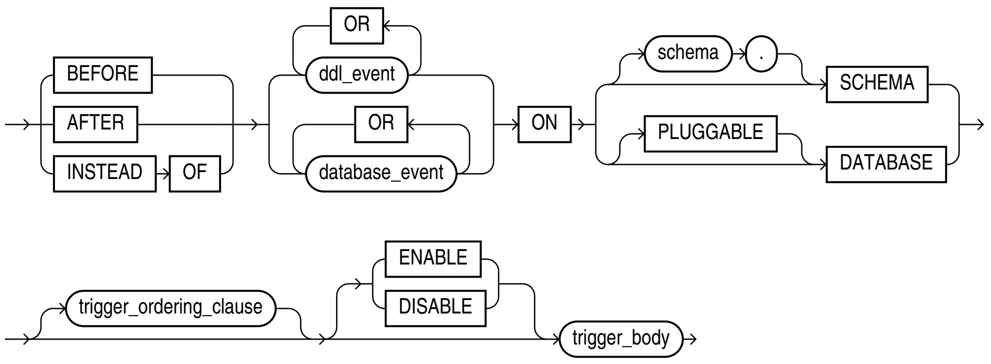
(*referencing_clause* ::= , *trigger_body* ::= , *trigger_edition_clause* ::= , *trigger_ordering_clause* ::=)

compound_dml_trigger **::=**



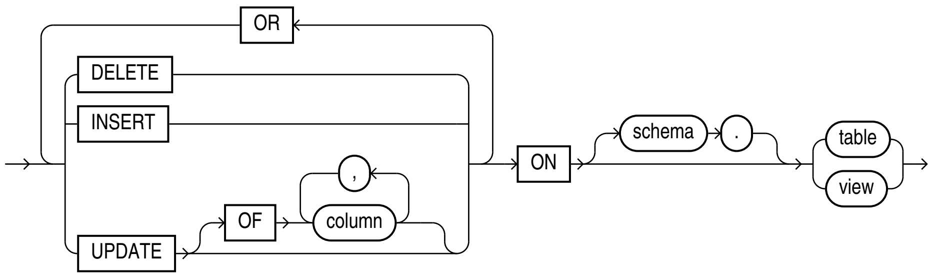
(*compound_trigger_block* ::= , *dml_event_clause* ::= , *referencing_clause* ::= , *trigger_edition_clause* ::= , *trigger_ordering_clause* ::=)

system_trigger **::=**

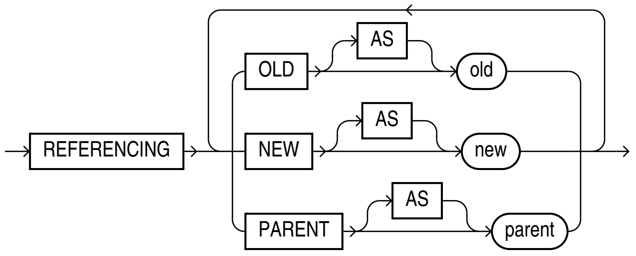


(*trigger_body* ::= , *trigger_ordering_clause* ::=)

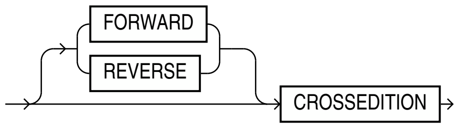
dml_event_clause **::=**



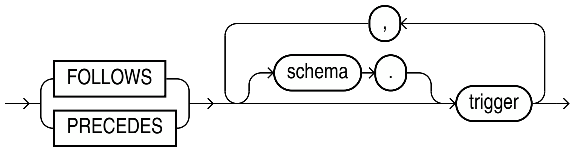
referencing_clause **::=**



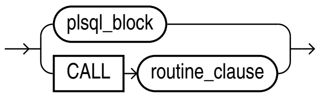
trigger_edition_clause **::=**



trigger_ordering_clause **::=**



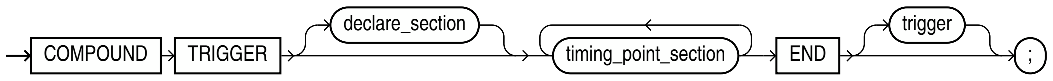
trigger_body **::=**



(*plsql_block ::=* ,

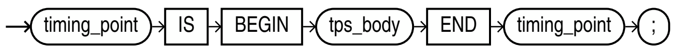
`routine_clause` in *Oracle Database SQL Language Reference*)

compound_trigger_block **::=**

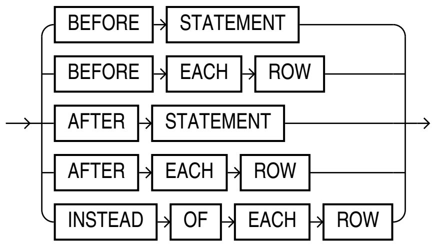


(*declare_section ::=*)

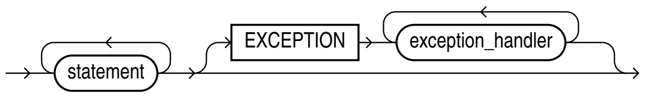
timing_point_section **::=**



timing_point **::=**



tps_body **::=**



(*exception_handler* ::= , *statement* ::=)

Semantics

create_trigger

OR REPLACE

Re-creates the trigger if it exists, and recompiles it.

Users who were granted privileges on the trigger before it was redefined can still access the procedure without being regranted the privileges.

[EDITIONABLE | NONEDITIONABLE]

Specifies whether the trigger is an editioned or noneditioned object if editioning is enabled for the schema object type `TRIGGER` in *`schema`*. **Default:** `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

Restriction on NONEDITIONABLE

You cannot specify `NONEDITIONABLE` for a crossedition trigger.

IF NOT EXISTS

Creates the trigger if it does not already exist. If a trigger by the same name does exist, the statement is ignored without error and the original trigger remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

Restrictions on *create_trigger*

See "Trigger Restrictions".

plsql_trigger_source

schema

Name of the schema for the trigger to be created. **Default:** your schema.

trigger

Name of the trigger to be created.

Triggers in the same schema cannot have the same names. Triggers can have the same names as other schema objects—for example, a table and a trigger can have the same name —however, to avoid confusion, this is not recommended.

If a trigger produces compilation errors, then it is still created, but it fails on execution. A trigger that fails on execution effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

Note:

If you create a trigger on a base table of a materialized view, then you must ensure that the trigger does not fire during a refresh of the materialized view. During refresh, the `DBMS_MVIEW` procedure `I_AM_A_REFRESH` returns `TRUE`.

simple_dml_trigger

Creates a simple DML trigger (described in "DML Triggers").

BEFORE

Causes the database to fire the trigger before running the triggering event. For row triggers, the trigger fires before each affected row is changed.

Restrictions on BEFORE

- You cannot specify a `BEFORE` trigger on a view unless it is an editioning view.
- In a `BEFORE` statement trigger, the trigger body cannot read `:NEW` or `:OLD`. (In a `BEFORE` row trigger, the trigger body can read and write the `:OLD` and `:NEW` fields.)

AFTER

Causes the database to fire the trigger after running the triggering event. For row triggers, the trigger fires after each affected row is changed.

Restrictions on AFTER

- You cannot specify an `AFTER` trigger on a view unless it is an editioning view.
- In an `AFTER` statement trigger, the trigger body cannot read `:NEW` or `:OLD`. (In an `AFTER` row trigger, the trigger body can read but not write the `:OLD` and `:NEW` fields.)

Note:

When you create a materialized view log for a table, the database implicitly creates an `AFTER` row trigger on the table. This trigger inserts a row into the materialized view log whenever an `INSERT`, `UPDATE`, or `DELETE` statement modifies data in the associated table. You cannot control the order in which multiple row triggers fire. Therefore, do not write triggers intended to affect the content of the materialized view.

See Also:

- *Oracle Database SQL Language Reference* for more information about materialized view logs
- *Oracle Database Development Guide* for information about editioning views

FOR EACH ROW

Creates the trigger as a row trigger. The database fires a row trigger for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the `WHEN` condition.

If you omit this clause, then the trigger is a statement trigger. The database fires a statement trigger only when the triggering statement is issued if the optional trigger constraint is met.

[ENABLE | DISABLE]

Creates the trigger in an enabled (default) or disabled state. Creating a trigger in a disabled state lets you ensure that the trigger compiles without errors before you enable it.

Note:

`DISABLE` is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors occur.

WHEN (*condition*)

Specifies a SQL condition that the database evaluates for each row that the triggering statement affects. If the value of *condition* is `TRUE` for an affected row, then *trigger_body* runs for that row; otherwise, *trigger_body* does not run for that row. The triggering statement runs regardless of the value of *condition*.

The *condition* can contain correlation names (see "*referencing_clause* ::=").

In *`condition`*, do not put a colon (:) before the correlation name `NEW`, `OLD`, or `PARENT` (in this context, it is not a placeholder for a bind variable).

See Also:

Oracle Database SQL Language Reference for information about SQL conditions

Restrictions on WHEN (*condition*)

- If you specify this clause, then you must also specify `FOR EACH ROW`.
- The *`condition`* cannot include a subquery or a PL/SQL expression (for example, an invocation of a user-defined function).

trigger_body

The PL/SQL block or `CALL` subprogram that the database runs to fire the trigger. A `CALL` subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If *trigger_body* is a PL/SQL block and it contains errors, then the `CREATE` [`OR REPLACE`] statement fails.

Restriction on *trigger_body*

The *declare_section* cannot declare variables of the data type `LONG` or `LONG RAW`.

instead_of_dml_trigger

Creates an `INSTEAD OF` DML trigger (described in "INSTEAD OF DML Triggers").

Restriction on INSTEAD OF

An `INSTEAD OF` trigger can read the `:OLD` and `:NEW` values, but cannot change them.

Note:

- If the view is inherently updatable and has `INSTEAD OF` triggers, the triggers take precedence: The database fires the triggers instead of performing DML on the view.
- If the view belongs to a hierarchy, then the subviews do not inherit the trigger.
- The `WITH CHECK OPTION` for views is not enforced when inserts or updates to the view are done using `INSTEAD OF` triggers. The `INSTEAD OF` trigger body must enforce the check. For information about `WITH CHECK OPTION`, see *Oracle Database SQL Language Reference*.
- The database fine-grained access control lets you define row-level security policies on views. These policies enforce specified rules in response to DML operations. If an `INSTEAD OF` trigger is also defined on the view, then the database does not enforce the row-level security policies, because the database fires the `INSTEAD OF` trigger instead of running the DML on the view.

DELETE

If the trigger is created on a noneditioning view, then `DELETE` causes the database to fire the trigger whenever a `DELETE` statement removes a row from the table on which the noneditioning view is defined.

If the trigger is created on a nested table column of a noneditioning view, then `DELETE` causes the database to fire the trigger whenever a `DELETE` statement removes an element from the nested table.

INSERT

If the trigger is created on a noneditioning view, then `INSERT` causes the database to fire the trigger whenever an `INSERT` statement adds a row to the table on which the noneditioning view is defined.

If the trigger is created on a nested table column of a noneditioning view, then `INSERT` causes the database to fire the trigger whenever an `INSERT` statement adds an element to the nested table.

UPDATE

If the trigger is created on a noneditioning view, then `UPDATE` causes the database to fire the trigger whenever an `UPDATE` statement changes a value in a column of the table on which the noneditioning view is defined.

If the trigger is created on a nested table column of a noneditioning view, then `UPDATE` causes the database to fire the trigger whenever an `UPDATE` statement changes a value in a column of the nested table.

nested_table_column

Name of the *`nested_table_column`* on which the trigger is to be created. The trigger fires only if the DML operates on the elements of the nested table. Performing DML operations directly on nested table columns does not cause the database to fire triggers defined on the table containing the nested table column. For more information, see "INSTEAD OF DML Triggers".

See Also:

`AS` *`subquery`* clause of `CREATE VIEW` in *Oracle Database SQL Language Reference* for a list of constructs that prevent inserts, updates, or deletes on a view

schema

Name of the schema containing the noneditioning view. **Default:** your schema.

noneditioning_view

If you specify *`nested_table_column`*, then *`noneditioning_view`* is the name of the noneditioning view that includes *`nested_table_column`*. Otherwise, *`noneditioning_view`* is the name of the noneditioning view on which the trigger is to be created.

FOR EACH ROW

For documentation only, because an `INSTEAD OF` trigger is always a row trigger.

`ENABLE`

(Default) Creates the trigger in an enabled state.

`DISABLE`

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

> **Note:**
>
> `DISABLE` is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors occur.

trigger_body

The PL/SQL block or `CALL` subprogram that the database runs to fire the trigger. A `CALL` subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If *trigger_body* is a PL/SQL block and it contains errors, then the `CREATE` [`OR REPLACE`] statement fails.

Restriction on *trigger_body*

The *`declare_section`* cannot declare variables of the data type `LONG` or `LONG RAW`.

compound_dml_trigger

Creates a compound DML trigger (described in "Compound DML Triggers").

ENABLE

(Default) Creates the trigger in an enabled state.

DISABLE

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

> **Note:**
>
> `DISABLE` is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors occur.

WHEN (*condition*)

Specifies a SQL condition that the database evaluates for each row that the triggering statement affects. If the value of *condition* is `TRUE` for an affected row, then *tps_body* runs for that row; otherwise, *tps_body* does not run for that row. The triggering statement runs regardless of the value of *condition*.

The *condition* can contain correlation names (see "*referencing_clause* ::="). In *condition*, do not put a colon (:) before the correlation name `NEW`, `OLD`, or `PARENT` (in this context, it is not a placeholder for a bind variable).

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about SQL conditions

Restrictions on WHEN (*condition*)

- If you specify this clause, then you must also specify at least one of these timing points:
  - `BEFORE EACH ROW`
  - `AFTER EACH ROW`
  - `INSTEAD OF EACH ROW`
- The *`condition`* cannot include a subquery or a PL/SQL expression (for example, an invocation of a user-defined function).

system_trigger

Defines a system trigger (described in "System Triggers").

BEFORE

Causes the database to fire the trigger before running the triggering event.

AFTER

Causes the database to fire the trigger after running the triggering event.

INSTEAD OF

Creates an `INSTEAD OF` trigger.

Restrictions on INSTEAD OF

- The triggering event must be a `CREATE` statement.
- You can create at most one `INSTEAD OF` DDL trigger (*non_dml_trigger*).

  For example, you can create an `INSTEAD OF` trigger on either the database or schema, but not on both the database and schema.

ddl_event

One or more types of DDL SQL statements that can cause the trigger to fire.

You can create triggers for these events on `DATABASE` or `SCHEMA` unless otherwise noted. You can create `BEFORE` and `AFTER` triggers for any of these events, but you can create `INSTEAD OF` triggers only for `CREATE` events. The database fires the trigger in the existing user transaction.

Note:

Some objects are created, altered, and dropped using PL/SQL APIs (for example, scheduler jobs are maintained by subprograms in the `DBMS_SCHEDULER` package). Such PL/SQL subprograms do not fire DDL triggers.

The following *ddl_event* values are valid:

- `ALTER`

  Causes the database to fire the trigger whenever an `ALTER` statement modifies a database object in the data dictionary. An `ALTER DATABASE` statement does not fire the trigger.
- `ANALYZE`

  Causes the database to fire the trigger whenever the database collects or deletes statistics or validates the structure of a database object.

See Also:

Oracle Database SQL Language Reference for information about using the SQL statement `ANALYZE` to collect statistics

- `ASSOCIATE STATISTICS`

  Causes the database to fire the trigger whenever the database associates a statistics type with a database object.

- `AUDIT`

  Causes the database to fire the trigger whenever an `AUDIT` statement is issued.

- `COMMENT`

  Causes the database to fire the trigger whenever a comment on a database object is added to the data dictionary.

- `CREATE`

  Causes the database to fire the trigger whenever a `CREATE` statement adds a database object to the data dictionary. The `CREATE DATABASE` or `CREATE CONTROLFILE` statement does not fire the trigger.

- `DISASSOCIATE STATISTICS`

  Causes the database to fire the trigger whenever the database disassociates a statistics type from a database object.

- `DROP`

  Causes the database to fire the trigger whenever a `DROP` statement removes a database object from the data dictionary.

- `GRANT`

  Causes the database to fire the trigger whenever a user grants system privileges or roles or object privileges to another user or to a role.

- `NOAUDIT`

  Causes the database to fire the trigger whenever a `NOAUDIT` statement is issued.

- `RENAME`

  Causes the database to fire the trigger whenever a `RENAME` statement changes the name of a database object.

- `REVOKE`

  Causes the database to fire the trigger whenever a `REVOKE` statement removes system privileges or roles or object privileges from a user or role.

- `TRUNCATE`

  Causes the database to fire the trigger whenever a `TRUNCATE` statement removes the rows from a table or cluster and resets its storage characteristics.

- `DDL`

  Causes the database to fire the trigger whenever any of the preceding DDL statements is issued.

database_event

One of the following database events. You can create triggers for these events on either `DATABASE` or `SCHEMA` unless otherwise noted. For each of these triggering events, the database opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction).

- `AFTER STARTUP`

  Causes the database to fire the trigger whenever the database is opened. This event is valid only with `DATABASE`, not with `SCHEMA`.

- `BEFORE SHUTDOWN`

  Causes the database to fire the trigger whenever an instance of the database is shut down. This event is valid only with `DATABASE`, not with `SCHEMA`.

- `AFTER DB_ROLE_CHANGE`

  In an Oracle Data Guard configuration, causes the database to fire the trigger whenever a role change occurs from standby to primary or from primary to standby. This event is valid only with `DATABASE`, not with `SCHEMA`.

  > **Note:**
  >
  > You cannot create an `AFTER DB_ROLE_CHANGE` trigger on a PDB.

- `AFTER SERVERERROR`

  Causes the database to fire the trigger whenever both of these conditions are true:

  – A server error message is logged.

  – Oracle relational database management system (RDBMS) determines that it is safe to fire error triggers.

    Examples of when it is unsafe to fire error triggers include:

    * RDBMS is starting up.
    * A critical error has occurred.

- `AFTER LOGON`

  Causes the database to fire the trigger whenever a client application logs onto the database.

- `BEFORE LOGOFF`

  Causes the database to fire the trigger whenever a client application logs off the database.

- `AFTER SUSPEND`

  Causes the database to fire the trigger whenever a server error causes a transaction to be suspended.

- `AFTER CLONE`

  Can be specified only if `PLUGGABLE DATABASE` is specified. After the PDB is copied (cloned), the database fires the trigger in the new PDB and then deletes the trigger. If the trigger fails, then the copy operation fails.

- `BEFORE UNPLUG`

Can be specified only if `PLUGGABLE DATABASE` is specified. Before the PDB is unplugged, the database fires the trigger and then deletes it. If the trigger fails, then the unplug operation fails.

- [`BEFORE` | `AFTER`] `SET CONTAINER`

  Causes the database to fire the trigger either before or after an `ALTER SESSION SET CONTAINER` statement runs.

See Also:

"Triggers for Publishing Events" for more information about responding to database events through triggers

[*schema*.]SCHEMA

Defines the trigger on the specified schema. **Default:** current schema. The trigger fires whenever any user connected as the specified schema initiates the triggering event.

[PLUGGABLE] DATABASE

`DATABASE` defines the trigger on the root. In a multitenant container database (CDB), only a common user who is connected to the root can create a trigger on the entire database.

`PLUGGABLE DATABASE` defines the trigger on the PDB to which you are connected.

The trigger fires whenever any user of the specified database or PDB initiates the triggering event.

Note:

If you are connected to a PDB, then specifying `DATABASE` is equivalent to specifying `PLUGGABLE DATABASE` unless you want to specify an option that applies only to a PDB (such as `CLONE` or `UNPLUG`).

ENABLE

(Default) Creates the trigger in an enabled state.

DISABLE

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

WHEN (*condition*)

Specifies a SQL condition that the database evaluates. If the value of *`condition`* is `TRUE`, then *`trigger_body`* runs for that row; otherwise, *`trigger_body`* does not run for that row. The triggering statement runs regardless of the value of *`condition`*.

See Also:

Oracle Database SQL Language Reference for information about SQL conditions

Restrictions on WHEN (*condition*)

- You cannot specify this clause for a `STARTUP`, `SHUTDOWN`, or `DB_ROLE_CHANGE` trigger.
- If you specify this clause for a `SERVERERROR` trigger, then *`condition`* must be `ERRNO =` *`error_code`*.
- The *`condition`* cannot include a subquery, a PL/SQL expression (for example, an invocation of a user-defined function), or a correlation name.

trigger_body

The PL/SQL block or `CALL` subprogram that the database runs to fire the trigger. A `CALL` subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If *`trigger_body`* is a PL/SQL block and it contains errors, then the `CREATE` [`OR REPLACE`] statement fails.

Restrictions on *trigger_body*

- The *`declare_section`* cannot declare variables of the data type `LONG` or `LONG RAW`.
- The trigger body cannot specify either `:NEW` or `:OLD`.

dml_event_clause

Specifies the triggering statements for *simple_dml_trigger* or *compound_dml_trigger*. The database fires the trigger in the existing user transaction.

DELETE

Causes the database to fire the trigger whenever a `DELETE` statement removes a row from *`table`* or the table on which *`view`* is defined.

INSERT

Causes the database to fire the trigger whenever an `INSERT` statement adds a row to *`table`* or the table on which *`view`* is defined.

UPDATE [OF *column* [, *column*]]

Causes the database to fire the trigger whenever an `UPDATE` statement changes a value in a specified column. **Default:** The database fires the trigger whenever an `UPDATE` statement changes a value in any column of *`table`* or the table on which *`view`* is defined.

If you specify a *`column`*, then you cannot change its value in the body of the trigger.

schema

Name of the schema that contains the database object on which the trigger is to be created. **Default:** your schema.

table

Name of the database table or object table on which the trigger is to be created.

Restriction on *schema.table*

You cannot create a trigger on a table in the schema `SYS`.

view

Name of the database view or object view on which the trigger is to be created.

> **Note:**
>
> A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section.

referencing_clause

Specifies correlation names, which refer to old, new, and parent values of the current row. **Defaults:** `OLD`, `NEW`, and `PARENT`.

If your trigger is associated with a table named `OLD`, `NEW`, or `PARENT`, then use this clause to specify different correlation names to avoid confusion between the table names and the correlation names.

If the trigger is defined on a nested table, then `OLD` and `NEW` refer to the current row of the nested table, and `PARENT` refers to the current row of the parent table. If the trigger is defined on a database table or view, then `OLD` and `NEW` refer to the current row of the database table or view, and `PARENT` is undefined.

Restriction on *referencing_clause*

The *referencing_clause* is not valid if *trigger_body* is `CALL` *routine*.

DML row-level triggers cannot reference fields of OLD/NEW/PARENT pseudorecords (correlation names) that correspond to columns with declared collation other than `USING_NLS_COMP`.

trigger_edition_clause

Creates the trigger as a crossedition trigger.

The handling of DML changes during edition-based redefinition (EBR) of an online application can entail multiple steps. Therefore, it is likely, though not required, that a crossedition trigger is also a **compound trigger**.

Restrictions on *trigger_edition_clause*

- You cannot define a crossedition trigger on a view.
- You cannot specify `NONEDITIONABLE` for a crossedition trigger.

FORWARD

(Default) Creates the trigger as a forward crossedition trigger. A forward crossedition trigger is intended to fire when DML changes are made in a database while an online application that uses the database is being patched or upgraded with EBR. The body of a crossedition trigger is designed to handle these DML changes so that they can be appropriately applied after the changes to the application code are completed.

REVERSE

Creates the trigger as a reverse crossedition trigger, which is intended to fire when the application, after being patched or upgraded with EBR, makes DML changes. This trigger propagates data to columns or tables used by the application before it was patched or upgraded.

> **See Also:**
>
> *Oracle Database Development Guide* for more information crossedition triggers

trigger_ordering_clause

FOLLOWS | PRECEDES

Specifies the relative firing of triggers that have the same timing point. It is especially useful when creating crossedition triggers, which must fire in a specific order to achieve their purpose.

Use `FOLLOWS` to indicate that the trigger being created must fire after the specified triggers. You can specify `FOLLOWS` for a conventional trigger or for a forward crossedition trigger.

Use `PRECEDES` to indicate that the trigger being created must fire before the specified triggers. You can specify `PRECEDES` only for a reverse crossedition trigger.

The specified triggers must exist, and they must have been successfully compiled. They need not be enabled.

If you are creating a noncrossedition trigger, then the specified triggers must be all of the following:

- Noncrossedition triggers
- Defined on the same table as the trigger being created
- Visible in the same edition as the trigger being created

If you are creating a crossedition trigger, then the specified triggers must be all of the following:

- Crossedition triggers
- Defined on the same table or editioning view as the trigger being created, unless you specify `FOLLOWS` or `PRECEDES`.

  If you specify `FOLLOWS`, then the specified triggers must be forward crossedition triggers, and if you specify `PRECEDES`, then the specified triggers must be reverse crossedition triggers. However, the specified triggers need not be on the same table or editioning view as the trigger being created.
- Visible in the same edition as the trigger being created

In the following definitions, A, B, C, and D are either noncrossedition triggers or forward crossedition triggers:

- If B specifies A in its `FOLLOWS` clause, then B **directly follows** A.
- If C directly follows B, and B directly follows A, then C **indirectly follows** A.
- If D directly follows C, and C indirectly follows A, then D indirectly follows A.
- If B directly or indirectly follows A, then B **explicitly follows** A (that is, the firing order of B and A is explicitly specified by one or more `FOLLOWS` clauses).

In the following definitions, A, B, C, and D are reverse crossedition triggers:

- If A specifies B in its `PRECEDES` clause, then A **directly precedes** B.

- If A directly precedes B, and B directly precedes C, then A **indirectly precedes** C.
- If A directly precedes B, and B indirectly precedes D, then A indirectly precedes D.
- If A directly or indirectly precedes B, then A **explicitly precedes** B (that is, the firing order of A and B is explicitly specified by one or more `PRECEDES` clauses).

Belongs to *compound_dml_trigger*.

compound_trigger_block

If the trigger is created on a noneditioning view, then *compound_trigger_block* must have only the `INSTEAD OF EACH ROW` section.

If the trigger is created on a table or editioning view, then timing point sections can be in any order, but no section can be repeated. The *compound_trigger_block* cannot have an `INSTEAD OF EACH ROW` section.

> **See Also:**
>
> "Compound DML Trigger Structure"

Restriction on *compound_trigger_block*

The *declare_section* of *compound_trigger_block* cannot include `PRAGMA AUTONOMOUS_TRANSACTION`.

> **See Also:**
>
> "Compound DML Trigger Restrictions"

timing_point

BEFORE STATEMENT

Specifies the `BEFORE STATEMENT` section of a *compound_dml_trigger* on a table or editioning view. This section causes the database to fire the trigger before running the triggering event.

Restriction on BEFORE STATEMENT

This section cannot specify `:NEW` or `:OLD`.

BEFORE EACH ROW

Specifies the `BEFORE EACH ROW` section of a *compound_dml_trigger* on a table or editioning view. This section causes the database to fire the trigger before running the triggering event. The trigger fires before each affected row is changed.

This section can read and write the `:OLD` and `:NEW` fields.

AFTER STATEMENT

Specifies the `AFTER STATEMENT` section of *compound_dml_trigger* on a table or editioning view. This section causes the database to fire the trigger after running the triggering event.

Restriction on AFTER STATEMENT

This section cannot specify `:NEW` or `:OLD`.

AFTER EACH ROW

Specifies the `AFTER EACH ROW` section of a *compound_dml_trigger* on a table or editioning view. This section causes the database to fire the trigger after running the triggering event. The trigger fires after each affected row is changed.

This section can read but not write the `:OLD` and `:NEW` fields.

INSTEAD OF EACH ROW

Specifies the `INSTEAD OF EACH ROW` section (the only timing point section) of a *compound_dml_trigger* on a noneditioning view. The database runs *tps_body* instead of running the triggering DML statement. For more information, see "INSTEAD OF DML Triggers".

Restriction on INSTEAD OF EACH ROW

- This section can appear only in a *`compound_dml_trigger`* on a noneditioning view.
- This section can read but not write the `:OLD` and `:NEW` values.

tps_body

The PL/SQL block or `CALL` subprogram that the database runs to fire the trigger. A `CALL` subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If *`tps_body`* is a PL/SQL block and it contains errors, then the `CREATE` [`OR REPLACE`] statement fails.

Restriction on *tps_body*

The *`declare_section`* cannot declare variables of the data type `LONG` or `LONG RAW`.

Examples

DML Triggers

- Example 10-1, "Trigger Uses Conditional Predicates to Detect Triggering Statement"
- Example 10-2, "INSTEAD OF Trigger"
- Example 10-3, "INSTEAD OF Trigger on Nested Table Column of View"
- Example 10-4, "Compound Trigger Logs Changes to One Table in Another Table"
- Example 10-5, "Compound Trigger Avoids Mutating-Table Error"

Triggers for Ensuring Referencial Integrity

- Example 10-6, "Foreign Key Trigger for Child Table"
- Example 10-7, "UPDATE and DELETE RESTRICT Trigger for Parent Table"
- Example 10-8, "UPDATE and DELETE SET NULL Trigger for Parent Table"
- Example 10-9, "DELETE CASCADE Trigger for Parent Table"
- Example 10-10, "UPDATE CASCADE Trigger for Parent Table"
- Example 10-11, "Trigger Checks Complex Constraints"
- Example 10-12, "Trigger Enforces Security Authorizations"
- Example 10-13, "Trigger Derives New Column Values"

Triggers That Use Correlation Names and Pseudorecords

- Example 10-14, "Trigger Logs Changes to EMPLOYEES.SALARY"
- Example 10-15, "Conditional Trigger Prints Salary Change Information"
- Example 10-16, "Trigger Modifies CLOB Columns"
- Example 10-17, "Trigger with REFERENCING Clause"
- Example 10-18, "Trigger References OBJECT_VALUE Pseudocolumn"

System Triggers

- Example 10-19, "BEFORE Statement Trigger on Sample Schema HR"
- Example 10-20, "AFTER Statement Trigger on Database"
- Example 10-21, "Trigger Monitors Logons"
- Example 10-22, "INSTEAD OF CREATE Trigger on Schema"

Miscellaneous Trigger Examples

- Example 10-23, "Trigger Invokes Java Subprogram"
- Example 10-24, "Trigger Cannot Handle Exception if Remote Database is Unavailable"
- Example 10-25, "Workaround for Trigger Cannot Handle Exception if Remote Database is Unavailable"
- Example 10-26, "Trigger Causes Mutating-Table Error"
- Example 10-27, "Update Cascade"
- Example 10-28, "Viewing Information About Triggers"

Related Topics

In this chapter:

- "ALTER TRIGGER Statement"
- "DROP TRIGGER Statement"

In other chapters:

- PL/SQL Triggers

See Also:

Oracle Database Development Guide for more information about crossedition triggers

CREATE TYPE Statement

The `CREATE TYPE` statement specifies the name of the type and its attributes, methods, and other properties.

The `CREATE TYPE` statement creates or replaces the specification of one of these:

- Abstract Data Type (ADT)
- Standalone varying array (varray) type

- Standalone nested table type
- Incomplete object type

  An **incomplete type** is a type created by a forward type definition. It is called incomplete because it has a name but no attributes or methods. It can be referenced by other types, allowing you to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

The `CREATE TYPE BODY` statement contains the code for the methods that implement the type.

Note:

- If you create a type whose specification declares only attributes but no methods, then you need not specify a type body.
- A standalone collection type that you create with the `CREATE TYPE` statement differs from a collection type that you define with the keyword `TYPE` in a PL/SQL block or package. For information about the latter, see "Collection Variable Declaration".
- With the `CREATE TYPE` statement, you can create nested table and `VARRAY` types, but not associative arrays. In a PL/SQL block or package, you can define all three collection types.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

To create a type in your schema, you must have the `CREATE TYPE` system privilege. To create a type in another user's schema, you must have the `CREATE ANY TYPE` system privilege. You can acquire these privileges explicitly or be granted them through a role.

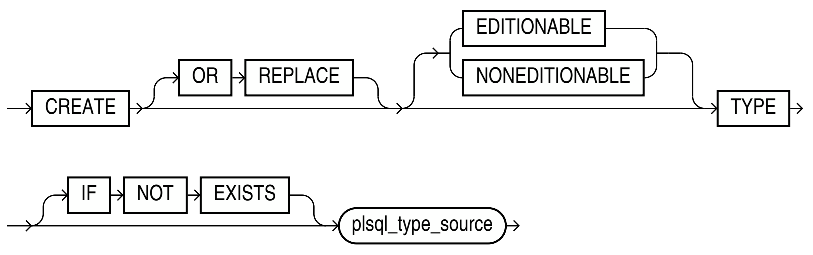
To create a subtype, you must have the `UNDER ANY TYPE` system privilege or the `UNDER` object privilege on the supertype.

The owner of the type must be explicitly granted the `EXECUTE` object privilege to access all other types referenced in the definition of the type, or the type owner must be granted the `EXECUTE ANY TYPE` system privilege. The owner cannot obtain these privileges through roles.

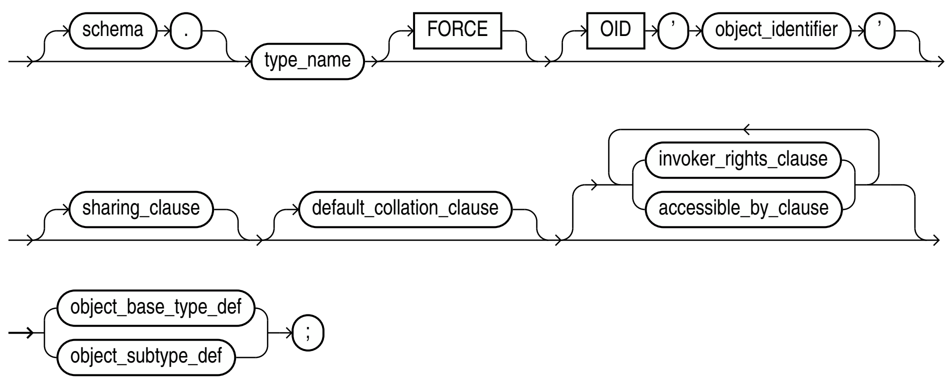
If the type owner intends to grant other users access to the type, then the owner must be granted the `EXECUTE` object privilege on the referenced types with the `GRANT OPTION` or the `EXECUTE ANY TYPE` system privilege with the `ADMIN OPTION`. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

Syntax

create_type **::=**

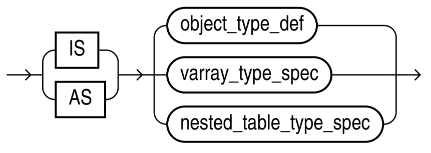


plsql_type_source **::=**



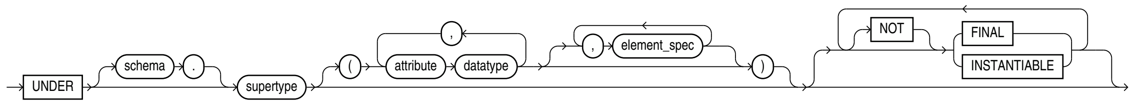
(*sharing_clause* ::=, *default_collation_clause* ::= , *accessible_by_clause* ::= , *invoker_rights_clause* ::= , *object_base_type_def* ::= , *object_subtype_def* ::=)

object_base_type_def **::=**



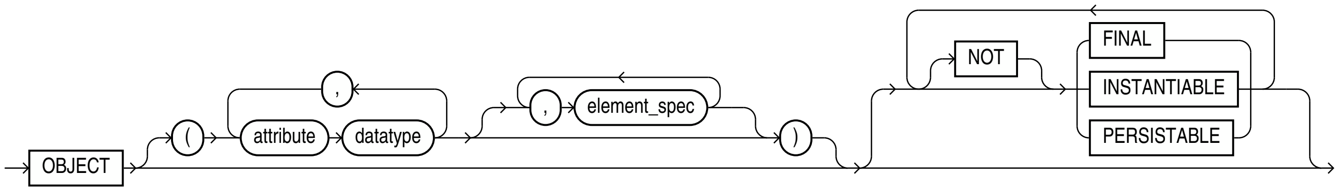
(*object_type_def* ::=, *nested_table_type_spec* ::=, *varray_type_spec* ::=)

object_subtype_def **::=**



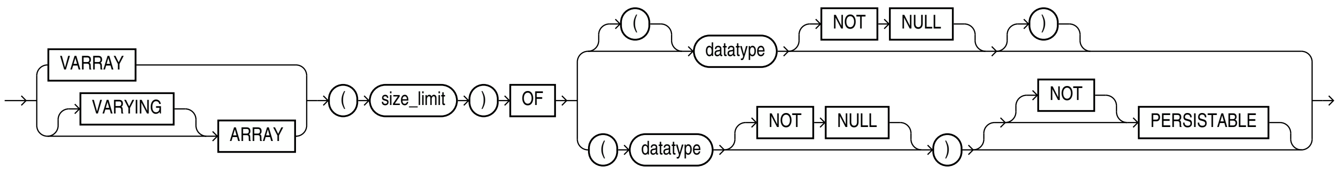
(*datatype* ::=, *element_spec* ::=)

object_type_def **::=**



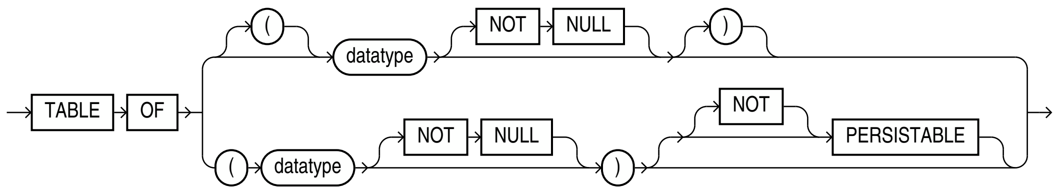
(*datatype* ::=, *element_spec* ::=)

varray_type_spec **::=**



(*datatype* ::=)

nested_table_type_spec **::=**



(*datatype* ::=)

Semantics

create_type

OR REPLACE

Re-creates the type if it exists, and recompiles it.

Users who were granted privileges on the type before it was redefined can still access the type without being regranted the privileges.

If any function-based indexes depend on the type, then the database marks the indexes `DISABLED`.

[EDITIONABLE | NONEDITIONABLE]

Specifies whether the type is an editioned or noneditioned object if editioning is enabled for the schema object type `TYPE` in *`schema`*. **Default:** `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

IF NOT EXISTS

Creates the type if it does not already exist. If a type by the same name does exist, the statement is ignored without error and the original type remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_type_source

schema

Name of the schema containing the type. **Default:** your schema.

type_name

Name of an ADT, a nested table type, or a `VARRAY` type.

If creating the type results in compilation errors, then the database returns an error. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

The database implicitly defines a constructor method for each user-defined type that you create. A **constructor** is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the name of the user-defined type. You can also create a user-defined constructor using the *`constructor_spec`* syntax.

The parameters of the ADT constructor method are the data attributes of the ADT. They occur in the same order as the attribute definition order for the ADT. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

FORCE

If *`type_name`* exists and has type dependents, but not table dependents, `FORCE` forces the statement to replace the type. (If *`type_name`* has table dependents, the statement fails with or without `FORCE`.)

Note:

If type `t1` has type dependent `t2`, and type `t2` has table dependents, then type `t1` also has table dependents.

See Also:

Oracle Database Object-Relational Developer's Guide

OID '*object_identifier*'

Establishes type equivalence of identical objects in multiple databases. See *Oracle Database Object-Relational Developer's Guide* for information about this clause.

object_base_type_def

Creates a schema-level ADT. Such ADTs are sometimes called **root** ADTs.

IS | AS

The keyword `IS` or `AS` is required when creating an ADT.

> **See Also:**
>
> "Example 15-25, ADT Examples"

object_subtype_def

Creates a subtype of an existing type.

UNDER *supertype*

The existing supertype must be an ADT. The subtype you create in this statement inherits the properties of its supertype. It must either override some of those properties or add properties to distinguish it from the supertype.

> **See Also:**
>
> "Example 15-26, Creating a Subtype" and "Example 15-27, Creating a Type Hierarchy"

attribute

Name of an ADT attribute. An ADT attribute is a data item with a name and a type specifier that forms the structure of the ADT. You must specify at least one attribute for each ADT. The name must be unique in the ADT, but can be used in other ADTs.

If you are creating a subtype, then the attribute name cannot be the same as any attribute or method name declared in the supertype chain.

datatype

The data type of an ADT attribute. This data type must be stored in the database; that is, either a predefined data type or a user-defined standalone collection type.

Restrictions on *datatype*

- You cannot impose the `NOT NULL` constraint on an attribute.
- You cannot specify attributes of type `ROWID`, `LONG`, or `LONG RAW`.
- You cannot specify a data type of `UROWID` for an ADT.
- If you specify an object of type `REF`, then the target object must have an object identifier.
- If you are creating a collection type for use as a nested table or varray column of a table, then you cannot specify attributes of type `ANYTYPE`, `ANYDATA`, or `ANYDATASET`.
- JSON cannot be an attribute of a user defined type (ADT).

object_type_def

Creates an ADT. The variables that form the data structure are called **attributes**. The member subprograms that define the behavior of the ADT are called **methods**.

OBJECT

The keyword `OBJECT` is required.

[NOT] FINAL, [NOT] INSTANTIABLE , [NOT] PERSISTABLE

At the schema level of the syntax, these clauses specify the inheritance attributes of the type.

[NOT] FINAL

Use the `[NOT] FINAL` clause to indicate whether any further subtypes can be created for this type:

- **(Default)** Specify `FINAL` if no further subtypes can be created for this type.
- Specify `NOT FINAL` if further subtypes can be created under this type.

[NOT] INSTANTIABLE

Use the `[NOT] INSTANTIABLE` clause to indicate whether any object instances of this type can be constructed:

- **(Default)** Specify `INSTANTIABLE` if object instances of this type can be constructed.
- Specify `NOT INSTANTIABLE` if no default or user-defined constructor exists for this ADT. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes, either inherited or specified in this statement.

[NOT] PERSISTABLE

Use `[NOT] PERSISTABLE` clause to indicate whether or not instances of the object type are persistable.

Only `PERSISTABLE` types can be stored in a table.

- **(Default)** You can specify `PERSISTABLE` if all the object type attributes are persistable. Creating a persistable object type with non-persistable attributes is not allowed.
- You can specify `NOT PERSISTABLE` if the object type attributes are persistable or non-persistable.
- Specify `NOT PERSISTABLE` if the ADT has a unique PL/SQL predefined type, such as SIMPLE_INTEGER and PLS_INTEGER.

Restrictions on [NOT] PERSISTABLE ADT

You cannot specify the `[NOT] PERSISTABLE` clause in a subtype definition. The persistance property of a subtype is inherited from its supertype.

Non-persistable ADTs with PL/SQL unique attributes are only allowed in the PL/SQL context.

See : Example 15-30, "Creating a Non-Persistable Object Type"

varray_type_spec

Creates the type as an ordered set of elements, each of which has the same data type.

Restrictions on *varray_type_spec*

You can create a `VARRAY` type of `XMLType` or of a LOB type for procedural purposes, for example, in PL/SQL or in view queries. However, database storage of such a varray is not supported, so you cannot create an object table or an column of such a `VARRAY` type.

See Also:

"Example 15-28, Creating a Varray Type"

[NOT] PERSISTABLE

(datatype [NOT NULL])

The parentheses before and after the `datatype` [`NOT NULL`] clause are required when `PERSISTABLE` is specified. The parentheses are optional if `PERSISTABLE` is not specified.

Use [`NOT`] `PERSISTABLE` clause to indicate whether or not instances of the collection type (`VARRAY` or nested table) are persistable.

- **(Default)** A collection can be `PERSISTABLE` only if the collection element type is persistable. Creating a persistable collection type with non-persistable element type is not allowed.
- Specify `NOT PERSISTABLE` if any element type of the collection is not persistable. You can specify `NOT PERSISTABLE` for any collection, whether the element type is persistable or not.
- Specify `NOT PERSISTABLE` if the collection has a unique PL/SQL predefined type, such as SIMPLE_INTEGER and PLS_INTEGER.

Restrictions on [NOT] PERSISTABLE Varray and Nested Array

Non-persistable types with PL/SQL unique attributes are only allowed in the PL/SQL context.

See Example 15-29, "Creating a Non-Persistable Nested Array" and Example 15-31, "Creating a Non-Persistable Varray"

nested_table_type_spec

Creates a named nested table of type *`datatype`*.

[NOT] PERSISTABLE

Same as for `VARRAY`, see " [NOT] PERSISTABLE"

See Also:

- "Example 15-32, Creating a Nested Table Type"
- "Example 15-33, Creating a Nested Table Type Containing a VARRAY"

Examples

Example 15-25 ADT Examples

This example shows how the sample type `customer_typ` was created for the sample Order Entry (`oe`) schema. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE customer_typ_demo AS OBJECT
    ( customer_id         NUMBER(6)
    , cust_first_name     VARCHAR2(20)
    , cust_last_name      VARCHAR2(20)
    , cust_address       CUST_ADDRESS_TYP
    , phone_numbers      PHONE_LIST_TYP
    , nls_language       VARCHAR2(3)
    , nls_territory      VARCHAR2(30)
    , credit_limit       NUMBER(9,2)
    , cust_email         VARCHAR2(30)
    , cust_orders        ORDER_LIST_TYP
    ) ;
/
```

In this example, the `data_typ1` ADT is created with one member function `prod`, which is implemented in the `CREATE TYPE BODY` statement:

```
CREATE TYPE data_typ1 AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );
/

CREATE TYPE BODY data_typ1 IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
         BEGIN
             RETURN (year + invent);
         END;
      END;
/
```

Example 15-26   Creating a Subtype

This statement shows how the subtype `corporate_customer_typ` in the sample `oe` schema was created.

It is based on the `customer_typ` supertype created in the preceding example and adds the `account_mgr_id` attribute. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE corporate_customer_typ_demo UNDER customer_typ
    ( account_mgr_id     NUMBER(6)
    );
/
```

Example 15-27   Creating a Type Hierarchy

These statements create a type hierarchy.

Type `employee_t` inherits the `name` and `ssn` attributes from type `person_t` and in addition has `department_id` and `salary` attributes. Type `part_time_emp_t` inherits all of the attributes from `employee_t` and, through `employee_t`, those of `person_t` and in addition has a `num_hrs` attribute. Type `part_time_emp_t` is final by default, so no further subtypes can be created under it.

```
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;
/
```

```
CREATE TYPE employee_t UNDER person_t
   (department_id NUMBER, salary NUMBER) NOT FINAL;
/

CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
/
```

You can use type hierarchies to create substitutable tables and tables with substitutable columns.

Example 15-28 Creating a Varray Type

This statement shows how the `phone_list_typ` `VARRAY` type with five elements in the sample `oe` schema was created.

A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE phone_list_typ_demo AS VARRAY(5) OF VARCHAR2(25);
/
```

Example 15-29 Creating a Non-Persistable Nested Array

This example shows how to create a PL/SQL nested array with unique PL/SQL predefined type `PLS_INTEGER` that is not persistable and can only be used in your PL/SQL programs.

```
CREATE TYPE IF NOT EXISTS varr_int AS VARRAY(10) OF (PLS_INTEGER) NOT
PERSISTABLE;
/
```

The optional `IF NOT EXISTS` clause is used to ensure that the statement is idempotent. The resulting output message (in this case `Type created`) is the same whether the type is created or the statement is ignored.

Example 15-30 Creating a Non-Persistable Object Type

This example shows how to create a PL/SQL object type with unique PL/SQL predefined type `PLS_INTEGER` that is not persistable and can only be used in your PL/SQL programs.

```
CREATE TYPE plsint AS OBJECT (I PLS_INTEGER) NOT PERSISTABLE;
/
```

Example 15-31 Creating a Non-Persistable Varray

This example shows how to create a PL/SQL varray with unique PL/SQL predefined type `PLS_INTEGER` that is not persistable and can only be used in your PL/SQL programs.

```
CREATE TYPE tab_plsint AS TABLE OF (PLS_INTEGER) NOT PERSISTABLE;
/
```

Example 15-32 Creating a Nested Table Type

This example from the sample schema `pm` creates the table type `textdoc_tab` of type `textdoc_typ`:

```
CREATE TYPE textdoc_typ AS OBJECT
    ( document_typ      VARCHAR2(32)
    , formatted_doc     BLOB
    ) ;
/

CREATE TYPE textdoc_tab AS TABLE OF textdoc_typ;
/
```

Example 15-33 Creating a Nested Table Type Containing a Varray

This example of multilevel collections is a variation of the sample table `oe.customers`.

In this example, the `cust_address` object column becomes a nested table column with the `phone_list_typ` varray column embedded in it. The phone_list_typ_demo type was created in "Example 15-28".

```
CREATE TYPE cust_address_typ2 AS OBJECT
       ( street_address     VARCHAR2(40)
       , postal_code        VARCHAR2(10)
       , city               VARCHAR2(30)
       , state_province     VARCHAR2(10)
       , country_id         CHAR(2)
       , phone              phone_list_typ_demo
       );
/

CREATE TYPE cust_nt_address_typ
   AS TABLE OF cust_address_typ2;
/
```

Example 15-34 Constructor Example

This example invokes the system-defined constructor to construct the `demo_typ` object and insert it into the `demo_tab` table.

```
CREATE TYPE demo_typ1 AS OBJECT (a1 NUMBER, a2 NUMBER);
/

CREATE TABLE demo_tab1 (b1 NUMBER, b2 demo_typ1);
/

INSERT INTO demo_tab1 VALUES (1, demo_typ1(2,3));
/
```

Example 15-35 Creating a Member Method

This example invokes method constructor `col.get_square`.

First the type is created:

```
CREATE TYPE demo_typ2 AS OBJECT (a1 NUMBER,
   MEMBER FUNCTION get_square RETURN NUMBER);
/
```

Next a table is created with an ADT column and some data is inserted into the table:

```
CREATE TABLE demo_tab2(col demo_typ2);
/

INSERT INTO demo_tab2 VALUES (demo_typ2(2));
/
```

The type body is created to define the member function, and the member method is invoked:

```
CREATE TYPE BODY demo_typ2 IS
   MEMBER FUNCTION get_square
   RETURN NUMBER
   IS x NUMBER;
   BEGIN
      SELECT c.col.a1*c.col.a1 INTO x
      FROM demo_tab2 c;
      RETURN (x);
   END;
END;
/

SELECT t.col.get_square() FROM demo_tab2 t;
/
```

Result:

```
T.COL.GET_SQUARE()
------------------
                 4
```

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

Example 15-36   Creating a Static Method

This example changes the definition of the `employee_t` type to associate it with the `construct_emp` function.

The example first creates an ADT `department_t` and then an ADT `employee_t` containing an attribute of type `department_t`:

```
CREATE OR REPLACE TYPE department_t AS OBJECT (
   deptno number(10),
   dname CHAR(30));
/

CREATE OR REPLACE TYPE employee_t AS OBJECT(
   empid RAW(16),
   ename CHAR(31),
   dept REF department_t,
      STATIC function construct_emp
      (name VARCHAR2, dept REF department_t)
      RETURN employee_t
);
/
```

This statement requires this type body statement.

```
CREATE OR REPLACE TYPE BODY employee_t IS
   STATIC FUNCTION construct_emp
   (name varchar2, dept REF department_t)
   RETURN employee_t IS
      BEGIN
         return employee_t(SYS_GUID(),name,dept);
      END;
END;
/
```

Next create an object table and insert into the table:

```
CREATE TABLE emptab OF employee_t;
/
INSERT INTO emptab
   VALUES (employee_t.construct_emp('John Smith', NULL));
/
```

Related Topics

- ALTER TYPE Statement
- CREATE TYPE BODY Statement
- DROP TYPE Statement
- Abstract Data Types
- Conditional Compilation Directive Restrictions
- Collection Variable Declaration
- Collection Types for information about user-defined standalone collection types
- PL/SQL Data Types
- *Oracle Database Object-Relational Developer's Guide* for more information about objects, incomplete types, varrays, and nested tables
- *Oracle Database Object-Relational Developer's Guide* for more information about constructors

CREATE TYPE BODY Statement

The CREATE TYPE BODY defines or implements the member methods defined in the type specification that was created with the CREATE TYPE statement.

For each method specified in a type specification for which you did not specify the *call_spec*, you must specify a corresponding method body in the type body.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

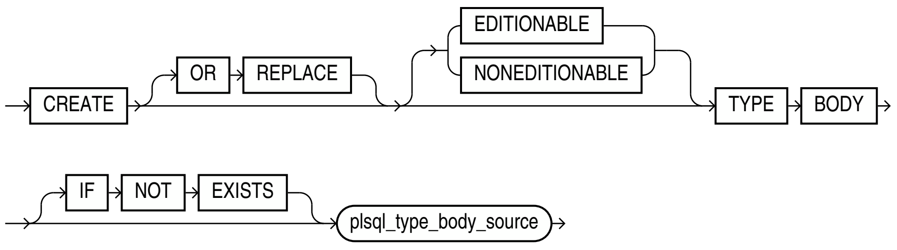
Prerequisites

Every member declaration in the CREATE TYPE specification for an ADT must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

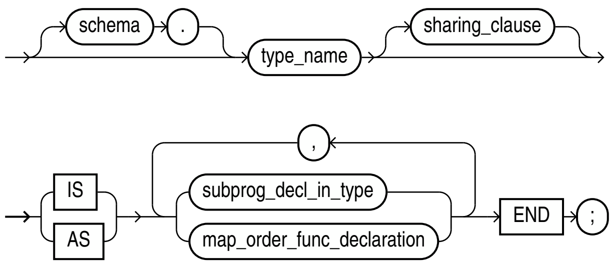
To create or replace a type body in your schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. To replace a type in another user's schema, you must have the DROP ANY TYPE system privilege.

Syntax

***create_type_body* ::=**

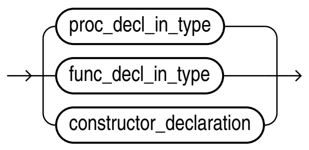


***plsql_type_body_source* ::=**

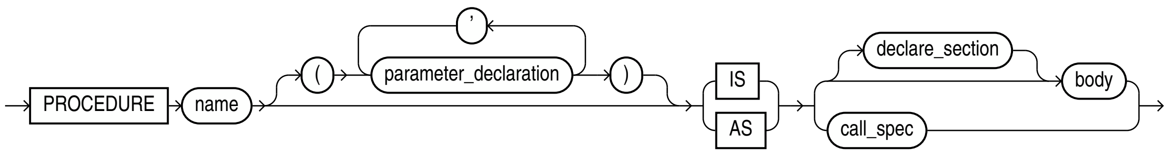


(*sharing_clause* ::=, *map_order_func_declaration* ::=, *subprog_decl_in_type* ::=)

***subprog_decl_in_type* ::=**

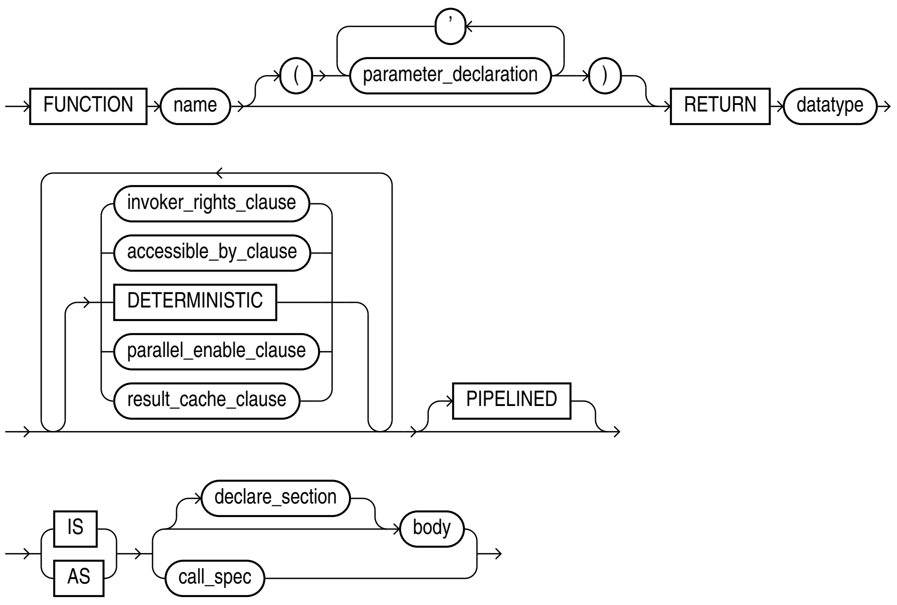


***proc_decl_in_type* ::=**



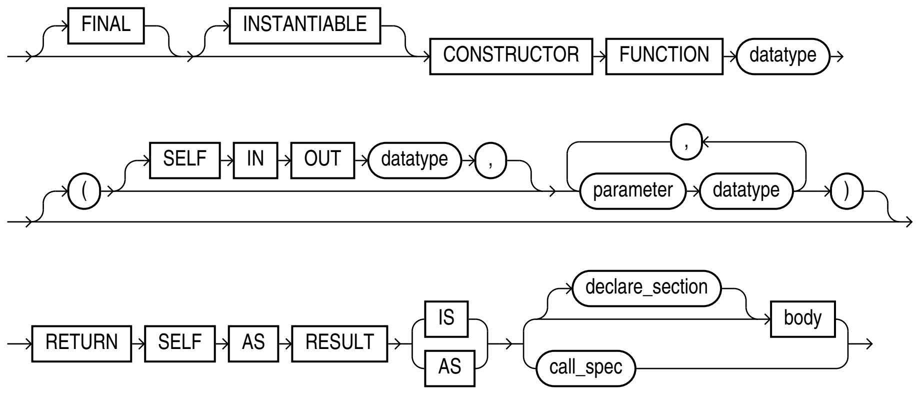
(*body* ::=, *call_spec* ::=, *declare_section* ::=, *parameter_declaration* ::=)

***func_decl_in_type* ::=**



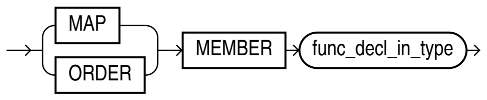
(*body* ::=, *invoker_rights_clause* ::=, *accessible_by_clause* ::=, *deterministic_clause* ::=, *call_spec* ::=, *declare_section* ::=, *parameter_declaration* ::=, *parallel_enable_clause* ::=, *result_cache_clause* ::=, *pipelined_clause* ::=)

constructor_declaration **::=**



(*call_spec* ::=)

map_order_func_declaration **::=**



Semantics

create_type_body

OR REPLACE

Re-creates the type body if it exists, and recompiles it.

Users who were granted privileges on the type body before it was redefined can still access the type body without being regranted the privileges.

You can use this clause to add member subprogram definitions to specifications added with the `ALTER TYPE` ... `REPLACE` statement.

[EDITIONABLE | NONEDITIONABLE]

If you do not specify this property, then the type body inherits `EDITIONABLE` or `NONEDITIONABLE` from the type specification. If you do specify this property, then it must match that of the type specification.

IF NOT EXISTS

Creates the type body if it does not already exist. If a type body by the same name does exist, the statement is ignored without error and the original type body remains unchanged.

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.

plsql_type_body_source

schema

Name of the schema containing the type body. **Default:** your schema.

type_name

Name of an ADT.

subprog_decl_in_type

The type of function or procedure subprogram associated with the type specification.

You must define a corresponding method name and optional parameter list in the type specification for each procedure or function declaration. For functions, you also must specify a return type.

map_order_func_declaration

You can declare either one `MAP` method or one `ORDER` method, regardless of how many `MEMBER` or `STATIC` methods you declare. If you declare either a `MAP` or `ORDER` 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.

MAP MEMBER

Declares or implements a `MAP` member function 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 specifies 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 type body can contain only one `MAP` method, which must be a function. The `MAP` function can have no arguments other than the implicit `SELF` argument.

ORDER MEMBER

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

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 database invokes the `ORDER MEMBER` *`func_decl_in_type`*.

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

proc_decl_in_type

A procedure subprogram declaration.

constructor_declaration

A user-defined constructor subprogram declaration. The `RETURN` clause of a constructor function must be `RETURN SELF AS RESULT`. This setting indicates that the most specific type of the value returned by the constructor function is the most specific type of the `SELF` argument that was passed in to the constructor function.

See Also:

- "CREATE TYPE Statement" for a list of restrictions on user-defined functions
- "Overloaded Subprograms" for information about overloading subprogram names
- *Oracle Database Object-Relational Developer's Guide* for information about and examples of user-defined constructors

declare_section

Declares items that are local to the procedure or function.

body

Procedure or function statements.

func_decl_in_type

A function subprogram declaration.

Examples

Several examples of creating type bodies appear in the Examples section of "CREATE TYPE Statement". For an example of re-creating a type body, see "Example 15-7".

Related Topics

- "CREATE TYPE Statement"
- "DROP TYPE BODY Statement"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"

DROP FUNCTION Statement

The `DROP FUNCTION` statement drops a standalone function from the database.

Note:

Do not use this statement to drop a function that is part of a package. Instead, either drop the entire package using the "DROP PACKAGE Statement" or redefine the package without the function using the "CREATE PACKAGE Statement" with the `OR REPLACE` clause.

Topics

- Prerequisites
- Syntax

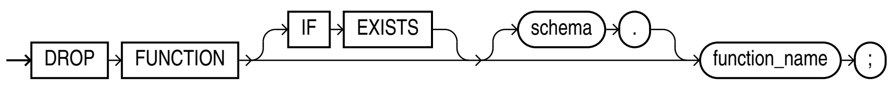
- Semantics
- Example
- Related Topics

Prerequisites

The function must be in your schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

***drop_function* ::=**



Semantics

drop_function

IF EXISTS

Drops the function if it exists. If no such function exists, the statement is ignored without error.

schema

Name of the schema containing the function. **Default:** your schema.

function_name

Name of the function to be dropped.

The database invalidates any local objects that depend on, or invoke, the dropped function. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped function.

If any statistics types are associated with the function, then the database disassociates the statistics types with the `FORCE` option and drops any user-defined statistics collected with the statistics type.

See Also:

- *Oracle Database SQL Language Reference* for information about the `ASSOCIATE STATISTICS` statement
- *Oracle Database SQL Language Reference* for information about the `DISASSOCIATE STATISTICS` statement

Example

Example 15-37 Dropping a Function

This statement drops the function `SecondMax` in the sample schema `oe` and invalidates all objects that depend upon `SecondMax`:

```
DROP FUNCTION IF EXISTS oe.SecondMax;
```

If `SecondMax` does not already exist in the schema, this statement is ignored without error. Note that the output message is the same whether or not the function exists (in this case, `Function dropped.`).

See Also:

"Example 15-15" for information about creating the `SecondMax` function

Related Topics

- "ALTER FUNCTION Statement"
- "CREATE FUNCTION Statement"

DROP LIBRARY Statement

The `DROP LIBRARY` statement drops an external procedure library from the database.

Topics

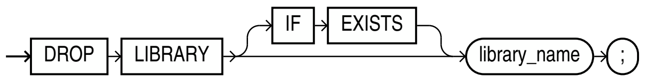
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

You must have the `DROP ANY LIBRARY` system privilege.

Syntax

drop_library ::=



Semantics

library_name

Name of the external procedure library being dropped.

IF EXISTS

Drops the library if it exists. If no such library exists, the statement is ignored without error.

Example

Example 15-38 Dropping a Library

The following statement drops the `ext_lib` library, which was created in "CREATE LIBRARY Statement":

```
DROP LIBRARY IF EXISTS ext_lib;
```

If `ext_lib` does not already exist, this statement is ignored without error. Note that the output message is the same whether or not the library exists (in this case, `Library dropped.`).

Related Topics

- "ALTER LIBRARY Statement"
- "CREATE LIBRARY Statement"

DROP PACKAGE Statement

The `DROP PACKAGE` statement drops a stored package from the database.

This statement drops the body and specification of a package.

Note:

Do not use this statement to drop a single object from a package. Instead, re-create the package without the object using the "CREATE PACKAGE Statement" and "CREATE PACKAGE BODY Statement" with the `OR REPLACE` clause.

Topics

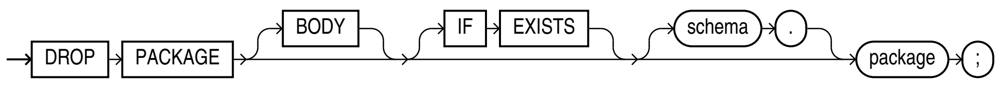
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The package must be in your schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

drop_package **::=**



Semantics

drop_package

BODY

Drops only the body of the package. If you omit this clause, then the database drops both the body and specification of the package.

When you drop only the body of a package but not its specification, the database does not invalidate dependent objects. However, you cannot invoke a procedure or stored function declared in the package specification until you re-create the package body.

IF EXISTS

Drops the package if it exists. If no such package exists, the statement is ignored without error.

schema

Name of the schema containing the package. **Default:** your schema.

package

Name of the package to be dropped.

The database invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, then the database disassociates the statistics types with the `FORCE` clause and drops any user-defined statistics collected with the statistics types.

See Also:

- *Oracle Database SQL Language Reference* for information about the `ASSOCIATE STATISTICS` statement
- *Oracle Database SQL Language Reference* for information about the `DISASSOCIATE STATISTICS` statement

Example

Example 15-39 Dropping a Package

This statement drops the specification and body of the `emp_mgmt` package, which was created in "CREATE PACKAGE BODY Statement", invalidating all objects that depend on the specification:

```
DROP PACKAGE emp_mgmt;
```

Related Topics

- "ALTER PACKAGE Statement"
- "CREATE PACKAGE Statement"
- "CREATE PACKAGE BODY Statement"

DROP PROCEDURE Statement

The `DROP PROCEDURE` statement drops a standalone procedure from the database.

> **Note:**
>
> Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the "DROP PACKAGE Statement", or redefine the package without the procedure using the "CREATE PACKAGE Statement" with the `OR REPLACE` clause.

Topics

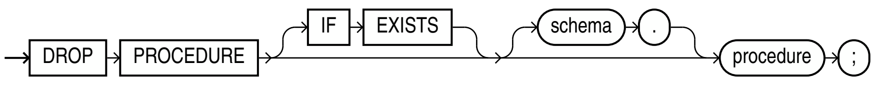
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The procedure must be in your schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

drop_procedure **::=**



Semantics

IF EXISTS

Drops the procedure if it exists. If no such procedure exists, the statement is ignored without error.

schema

Name of the schema containing the procedure. **Default:** your schema.

procedure

Name of the procedure to be dropped.

When you drop a procedure, the database invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

Example

Example 15-40   Dropping a Procedure

This statement drops the procedure `remove_emp` owned by the user `hr` and invalidates all objects that depend upon `remove_emp`:

```
DROP PROCEDURE IF EXISTS hr.remove_emp;
```

If `remove_emp` does not already exist in the schema, this statement is ignored without error. Note that the output message is the same whether or not the procedure exists (in this case, `Procedure dropped.`).

Related Topics

- "ALTER PROCEDURE Statement"
- "CREATE PROCEDURE Statement"

DROP TRIGGER Statement

The `DROP TRIGGER` statement drops a database trigger from the database.

Topics

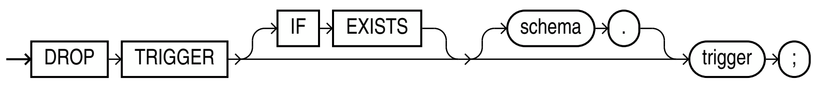
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The trigger must be in your schema or you must have the `DROP ANY TRIGGER` system privilege. To drop a trigger on `DATABASE` in another user's schema, you must also have the `ADMINISTER DATABASE TRIGGER` system privilege.

Syntax

drop_trigger **::=**



Semantics

IF EXISTS

Drops the trigger if it exists. If no such trigger exists, the statement is ignored without error.

schema

Name of the schema containing the trigger. **Default:** your schema.

trigger

Name of the trigger to be dropped.

Example

Example 15-41 Dropping a Trigger

This statement drops the `salary_check` trigger in the schema `hr`:

```
DROP TRIGGER hr.salary_check;
```

Related Topics

- "ALTER TRIGGER Statement"
- "CREATE TRIGGER Statement"

DROP TYPE Statement

The `DROP TYPE` statement drops the specification and body of an ADT, `VARRAY` type, or nested table type.

Topics

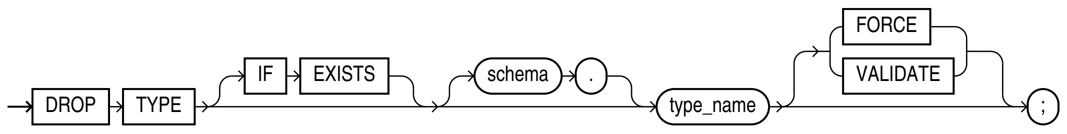
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The ADT, `VARRAY` type, or nested table type must be in your schema or you must have the `DROP ANY TYPE` system privilege.

Syntax

drop_type **::=**



Semantics

IF EXISTS

Drops the type if it exists. If no such type exists, the statement is ignored without error.

schema

Name of the schema containing the type. **Default:** your schema.

type_name

Name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If `type_name` is a supertype, then this statement fails unless you also specify `FORCE`. If you specify `FORCE`, then the database invalidates all subtypes depending on this supertype.

If `type_name` is a statistics type, then this statement fails unless you also specify `FORCE`. If you specify `FORCE`, then the database first disassociates all objects that are associated with `type_name` and then drops `type_name`.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the `ASSOCIATE STATISTICS` statement
> - *Oracle Database SQL Language Reference* for information about the `DISASSOCIATE STATISTICS` statement

If `type_name` is an ADT that is associated with a statistics type, then the database first tries to disassociate `type_name` from the statistics type and then drops `type_name`. However, if statistics have been collected using the statistics type, then the database cannot disassociate `type_name` from the statistics type, and this statement fails.

If `type_name` is an implementation type for an index type, then the index type is marked `INVALID`.

If `type_name` has a public synonym defined on it, then the database also drops the synonym.

Unless you specify `FORCE`, you can drop only types that are standalone schema objects with no dependencies. This is the default behavior.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about the `CREATE INDEXTYPE` statement

FORCE

Drops the type even if it has dependent database objects. The database marks `UNUSED` all columns dependent on the type to be dropped, and those columns become inaccessible.

> **Note:**
>
> Oracle recommends against specifying `FORCE` to drop object types with dependencies. This operation is not recoverable and might make the data in the dependent tables or columns inaccessible.

VALIDATE

Causes the database to check for stored instances of this type in substitutable columns of any of its supertypes. If no such instances are found, then the database completes the drop operation.

This clause is meaningful only for subtypes. Oracle recommends the use of this option to safely drop subtypes that do not have any explicit type or table dependencies.

Example

Example 15-42 Dropping an ADT

This statement removes the ADT `person_t`. See "CREATE TYPE Statement" for the example that creates this ADT. Any columns that are dependent on `person_t` are marked `UNUSED` and become inaccessible.

```
DROP TYPE IF EXISTS person_t FORCE;
```

If `person_t` does not already exist, this statement is ignored without error. Note that the output message is the same whether or not the ADT exists (`Type dropped.`).

Related Topics

- "ALTER TYPE Statement"
- "CREATE TYPE Statement"
- "CREATE TYPE BODY Statement"

DROP TYPE BODY Statement

The `DROP TYPE BODY` statement drops the body of an ADT, `VARRAY` type, or nested table type.

When you drop a type body, the type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the type, although you cannot invoke its member functions.

Topics

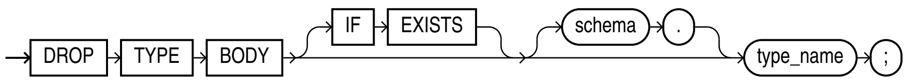
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The type body must be in your schema or you must have the `DROP ANY TYPE` system privilege.

Syntax

drop_type_body **::=**



Semantics

IF EXISTS

Drops the type body if it exists. If no such type body exists, the statement is ignored without error.

schema

Name of the schema containing the type. **Default:** your schema.

type_name

Name of the type body to be dropped.

Restriction on *type_name*

You can drop a type body only if it has no type or table dependencies.

Example

Example 15-43 Dropping an ADT Body

This statement removes the ADT body `data_typ1`. See "CREATE TYPE Statement" for the example that creates this ADT.

```
DROP TYPE BODY data_typ1;
```

Related Topics

- "ALTER TYPE Statement"
- "CREATE TYPE Statement"
- "CREATE TYPE BODY Statement"