CREATE TYPE

Purpose

Use the `CREATE TYPE` statement to create the specification of an **object type**, a **SQLJ object type**, a named varying array (**varray**), a **nested table type**, or an **incomplete object type**. You create object types with the `CREATE TYPE` and the `CREATE TYPE BODY` statements. The `CREATE TYPE` statement specifies the name of the object type, its attributes, methods, and other properties. The `CREATE TYPE BODY` statement contains the code for the methods that implement the type.

Notes:

- If you create an object type for which the type specification declares only attributes but no methods, then you need not specify a type body.
- If you create a SQLJ object type, then you cannot specify a type body. The implementation of the type is specified as a Java class.

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, and so can be used 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.

See Also:

- CREATE TYPE BODY for information on creating the member methods of a type
- *PL/SQL User's Guide and Reference*, *Oracle Database Application Developer's Guide - Object-Relational Features*, and *Oracle Database Concepts* for more information about objects, incomplete types, varrays, and nested tables

---

Prerequisites

To create a type in your own 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.

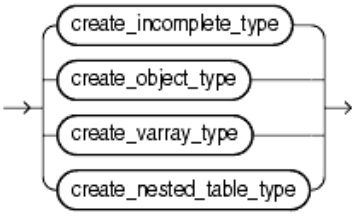
To create a subtype, you must have the `UNDER ANY TYPE` system privilege or the `UNDER` object privilege on the supertype.

The owner of the type must be explicitly granted the `EXECUTE` object privilege in order to access all other types referenced within 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

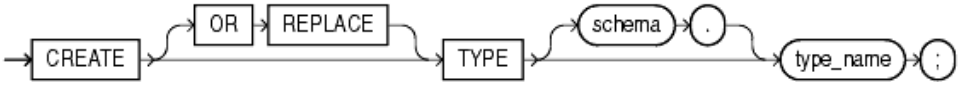
create_type::=



Description of the illustration create_type.gif

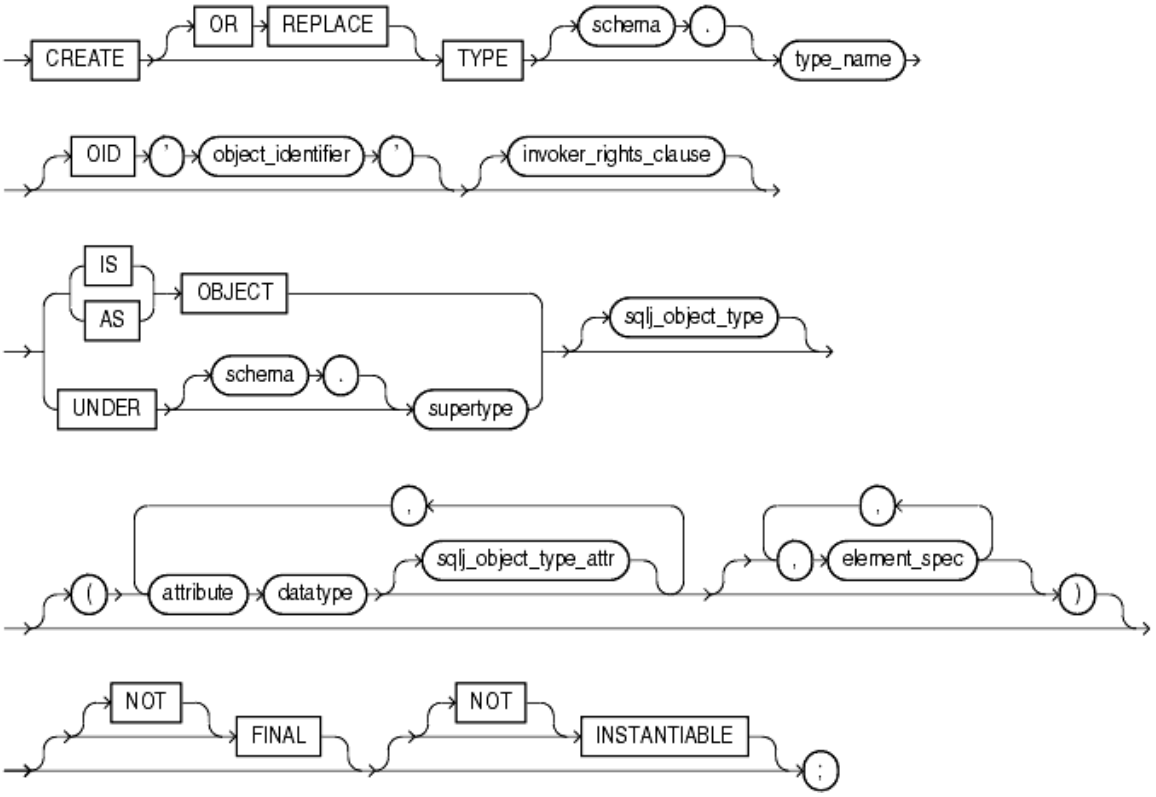
(*create_incomplete_type::=*, *create_object_type::=*, *create_varray_type::=*, *create_nested_table_type::=*)

create_incomplete_type::=



Description of the illustration create_incomplete_type.gif

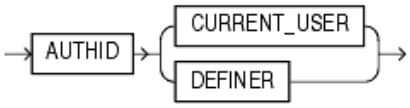
***create_object_type*::=**



Description of the illustration create_object_type.gif

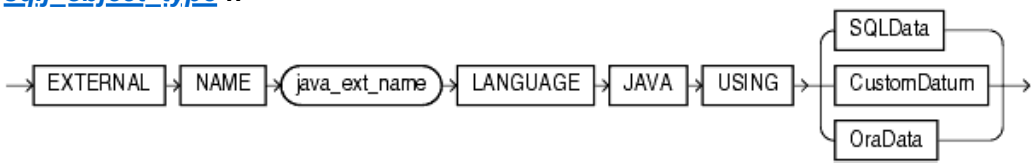
(*invoker_rights_clause::=*, *element_spec::=*)

***invoker_rights_clause*::=**



Description of the illustration invoker_rights_clause.gif

***sqlj_object_type* ::=**

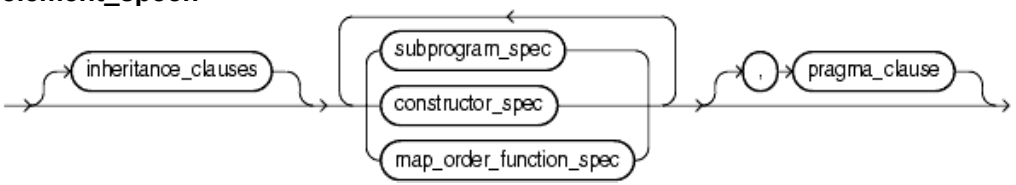


Description of the illustration sqlj_object_type.gif

***sqlj_object_type_attr* ::=**

Description of the illustration sqlj_object_type_attr.gif

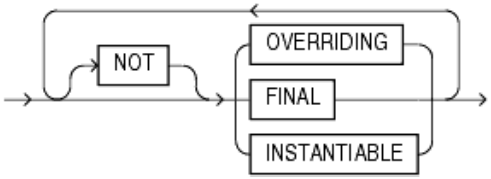
element_spec::=



Description of the illustration element_spec.gif

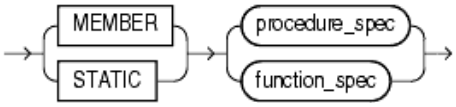
(*inheritance_clauses::=*, *subprogram_spec::=*, *constructor_spec ::=*, *map_order_function_spec::=*, *pragma_clause ::=*)

inheritance_clauses::=



Description of the illustration inheritance_clauses.gif

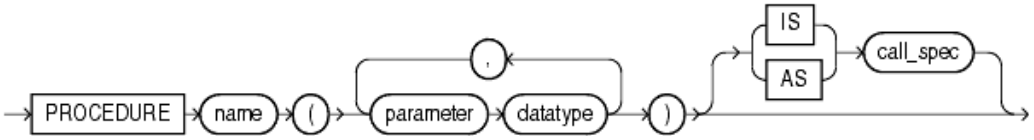
subprogram_spec::=



Description of the illustration subprogram_spec.gif

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

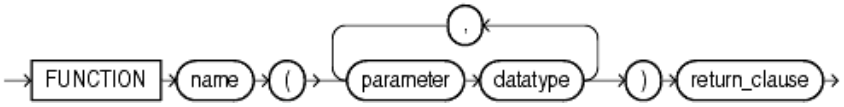
procedure_spec::=



Description of the illustration procedure_spec.gif

(*call_spec ::=*)

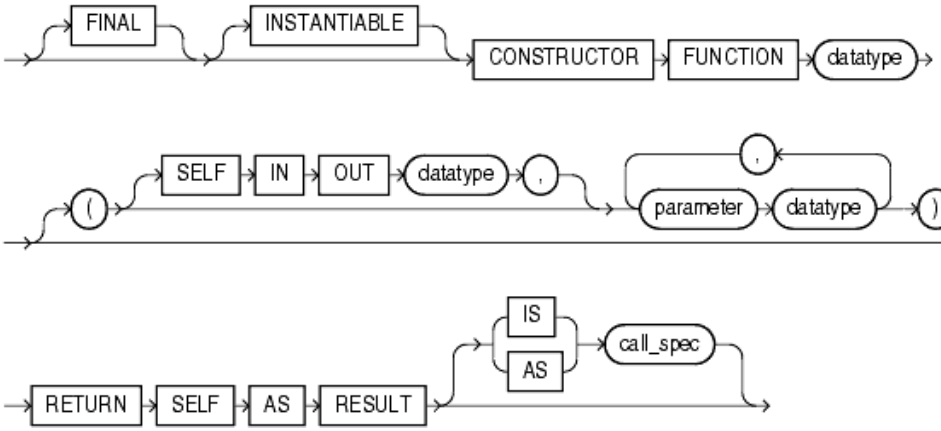
function_spec::=



Description of the illustration function_spec.gif

(*return_clause::=*)

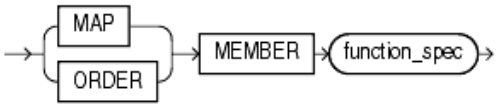
constructor_spec **::=**



Description of the illustration constructor_spec.gif

(*call_spec ::=*)

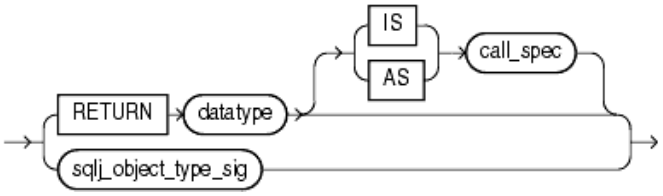
map_order_function_spec::=



Description of the illustration map_order_function_spec.gif

(*function_spec::=*)

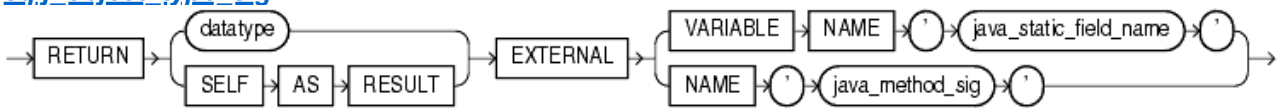
***return_clause*::=**



Description of the illustration return_clause.gif

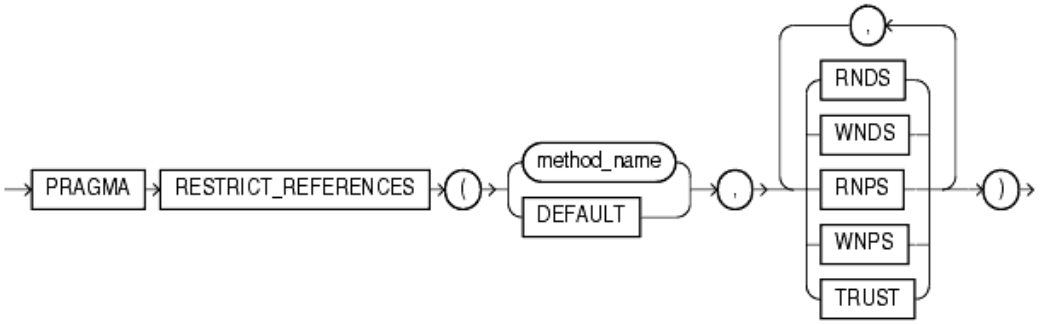
(*call_spec ::=*, *sqlj_object_type_sig::=*)

***sqlj_object_type_sig*::=**



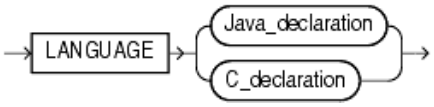
Description of the illustration sqlj_object_type_sig.gif

***pragma_clause* ::=**



Description of the illustration pragma_clause.gif

***call_spec* ::=**

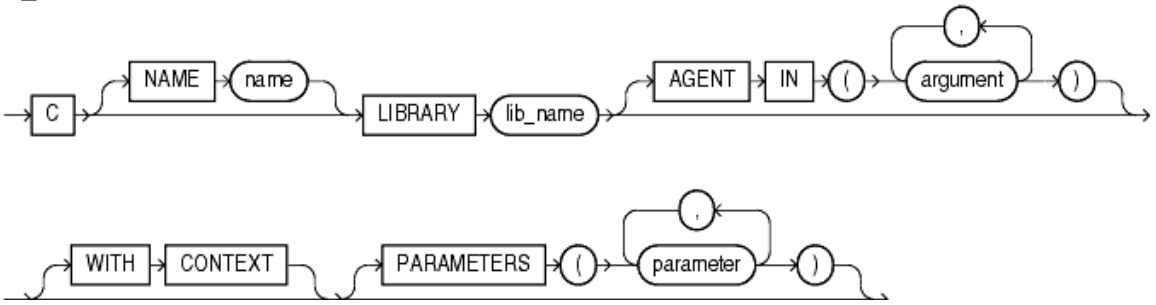


Description of the illustration call_spec.gif

Java_declaration::=

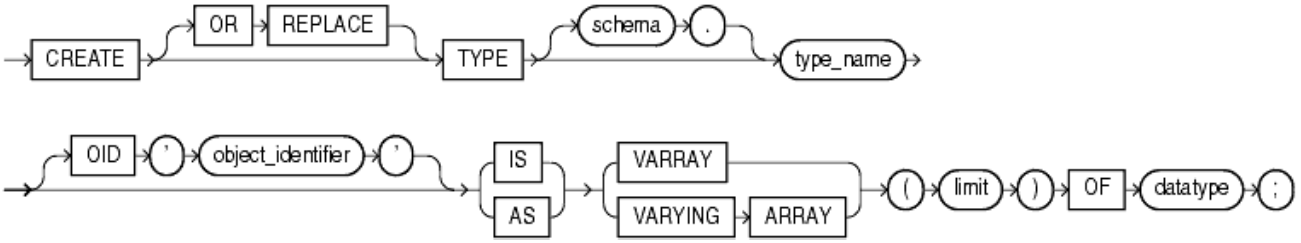
Description of the illustration Java_declaration.gif

C_declaration::=



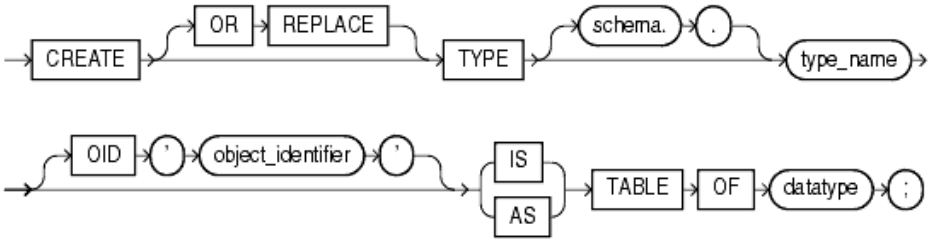
Description of the illustration C_declaration.gif

***create_varray_type*::=**



Description of the illustration create_varray_type.gif

***create_nested_table_type*::=**



Description of the illustration create_nested_table_type.gif

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the type if it already exists. Use this clause to change the definition of an existing type without first dropping it.

Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again.

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

schema

Specify the schema to contain the type. If you omit *`schema`*, then Oracle Database creates the type in your current schema.

type_name

Specify the name of an object type, 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`.

Oracle 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 same as the name of the user-defined type. You can also create a user-defined constructor using the *`constructor_spec`* syntax.

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

create_object_type

Use the *`create_object_type`* clause to create a user-defined object type. The variables that form the data structure are called **attributes**. The member subprograms that define the behavior of the object are called **methods**. The keywords `AS OBJECT` are required when creating an object type.

See Also:

"Object Type Examples"

OID Clause

The `OID` clause is useful for establishing type equivalence of identical objects in more than one database. Please refer to *Oracle Data Cartridge Developer's Guide* for information on this clause.

invoker_rights_clause

The *`invoker_rights_clause`* lets you specify whether the member functions and procedures of the object type execute with the privileges and in the schema of the user who owns the object type or with the privileges and in the schema of `CURRENT_USER`. This specification applies to the corresponding type body as well.

This clause also determines how Oracle Database resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

- Specify `AUTHID CURRENT_USER` if you want the member functions and procedures of the object type to execute with the privileges of `CURRENT_USER`. This clause creates an **invoker-rights type**.

  This clause also indicates that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of `CURRENT_USER`. External names in all other statements resolve in the schema in which the type resides.

- Specify `AUTHID DEFINER` if you want the member functions and procedures of the object type to execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default and creates a **definer-rights type**.

Restrictions on Invoker Rights

- You can specify this clause only for an object type, not for a nested table or varray type.
- You can specify this clause for clarity if you are creating a subtype. However, subtypes inherit the rights model of their supertypes, so you cannot specify a different value than was specified for the supertype.
- If the supertype was created with definer's rights, then you must create the subtype in the same schema as the supertype.

See Also:

- *Oracle Database Concepts* and *Oracle Database Application Developer's Guide - Fundamentals* for information on how `CURRENT_USER` is determined
- *Oracle Database Security Guide* for information on invoker-rights and definer-rights types

AS OBJECT Clause

Specify `AS OBJECT` to create a top-level object type. Such object types are sometimes called **root** object types.

UNDER Clause

Specify `UNDER` *`supertype`* to create a subtype of an existing type. The existing supertype must be an object type. The subtype you create in this statement inherits the properties of its supertype. It must either override some of those properties or add new properties to distinguish it from the supertype.

See Also:

"Subtype Example" and "Type Hierarchy Example"

sqlj_object_type

Specify this clause to create a **SQLJ object type**. In a SQLJ object type, you map a Java class to a SQL user-defined type. You can then define tables or columns on the SQLJ object type as you would with any other user-defined type.

You can map one Java class to multiple SQLJ object types. If there exists a subtype or supertype of a SQLJ object type, then it must also be a SQLJ object type. That is, all types in the hierarchy must be SQLJ object types.

java_ext_name

Specify the name of the Java class. If the class exists, it must be public. The Java external name, including the schema, will be validated.

Multiple SQLJ object types can be mapped to the same class. However:

- A subtype must be mapped to a class that is an immediate subclass of the class to which its supertype is mapped.
- Two subtypes of a common supertype cannot be mapped to the same class.

SQLData | CustomDatum | OraData

Choose the mechanism for creating the Java instance of the type. `SQLData`, `CustomDatum`, and `OraData` are the interfaces that determine which mechanism will be used.

See Also:

Oracle Database JDBC Developer's Guide and Reference for information on these three interfaces and "SQLJ Object Type Example"

element_spec

The *`element_spec`* lets you specify each attribute of the object type.

attribute

For *attribute*, specify the name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object. You must specify at least one attribute for each object type.

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

datatype

For *datatype*, specify the Oracle Database built-in datatype or user-defined type of the attribute.

Restrictions on Attribute Datatypes

- You cannot specify attributes of type `ROWID`, `LONG`, or `LONG RAW`.
- You cannot specify a datatype of `UROWID` for a user-defined object type.
- 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`.

See Also:

"Datatypes " for a list of valid datatypes

---

sqlj_object_type_attr

This clause is valid only if you have specified the *sqlj_object_type* clause—that is, you are mapping a Java class to a SQLJ object type. Specify the external name of the Java field that corresponds to the attribute of the SQLJ object type. The Java *field_name* must already exist in the class. You cannot map a Java *field_name* to more than one SQLJ object type attribute in the same type hierarchy.

This clause is optional when you create a SQLJ object type.

subprogram_spec

The *subprogram_spec* lets you associate a procedure subprogram with the object type.

MEMBER Clause

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

Restriction on Member Methods

You cannot specify a `MEMBER` method if you are mapping a Java class to a SQLJ object type.

See Also:

"Creating a Member Method: Example"

STATIC Clause

Specify a function or procedure subprogram associated with the object type. Unlike MEMBER methods, STATIC methods do not have any implicit parameters. That is, you cannot reference SELF in their body. They are typically invoked as *type_name.method*().

Restrictions on Static Methods

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

See Also:

"Creating a Static Method: Example"

[NOT] FINAL, [NOT] INSTANTIABLE

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

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

- Specify FINAL if no further subtypes can be created for this type. This is the default.
- Specify NOT FINAL if further subtypes can be created under this type.

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

- Specify INSTANTIABLE if object instances of this type can be constructed. This is the default.
- Specify NOT INSTANTIABLE if no default or user-defined constructor exists for this object 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.

inheritance_clauses

As part of the *element_spec*, the *inheritance_clauses* let you specify the relationship between supertypes and subtypes.

OVERRIDING

This clause is valid only for MEMBER methods. Specify OVERRIDING to indicate that this method overrides a MEMBER method defined in the supertype. This keyword is required if the method redefines a supertype method. NOT OVERRIDING is the default.

Restriction on OVERRIDING

The OVERRIDING clause is not valid for a STATIC method or for a SQLJ object type.

FINAL

Specify FINAL to indicate that this method cannot be overridden by any subtype of this type. The default is NOT FINAL.

NOT INSTANTIABLE

Specify NOT INSTANTIABLE if the type does not provide an implementation for this method. By default all methods are INSTANTIABLE.

Restriction on NOT INSTANTIABLE

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

See Also:

constructor_spec

procedure_spec or *function_spec*

Use these clauses to specify the parameters and datatypes of the procedure or function. If this subprogram does not include the declaration of the procedure or function, then you must issue a corresponding `CREATE TYPE BODY` statement.

Restriction on Procedure and Function Specification

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

return_clause

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

See Also:

- *PL/SQL User's Guide and Reference* for information about method invocation and methods
- CREATE PROCEDURE and CREATE FUNCTION for the full syntax with all possible clauses
- CREATE TYPE BODY
- "Restrictions on User-Defined Functions" for a list of restrictions on user-defined functions

sqlj_object_type_sig

Use this form of the *return_clause* if you intend to create SQLJ object type functions or procedures.

- If you are mapping a Java class to a SQLJ object type and you specify `EXTERNAL NAME`, then the value of the Java method returned must be compatible with the SQL returned value, and the Java method must be public. Also, the method signature (method name plus parameter types) must be unique within the type hierarchy.
- If you specify `EXTERNAL VARIABLE NAME`, then the type of the Java static field must be compatible with the return type.

call_spec

Specify the call specification (call spec) that maps a Java or C method name, parameter types, and return type to their SQL counterparts. If all the member methods in the type have been defined in this clause, then you need not issue a corresponding `CREATE TYPE BODY` statement.

The *Java_declaration* string identifies the Java implementation of the method.

See Also:

Oracle Database Java Developer's Guide and *Oracle Database Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the Java and C declarations, respectively

pragma_clause

The *pragma_clause* lets you specify a compiler directive. The PRAGMA RESTRICT_REFERENCES compiler directive denies member functions read/write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

Note:

Oracle recommends that you avoid using this clause unless you must do so for backward compatibility of your applications. This clause has been deprecated, because Oracle Database now runs purity checks at run time.

---

method

Specify the name of the MEMBER function or procedure to which the pragma is being applied.

DEFAULT

Specify DEFAULT if you want the database to apply the pragma to all methods in the type for which a pragma has not been explicitly specified.

WNDS

Specify WNDS to enforce the constraint writes no database state, which means that the method does not modify database tables.

WNPS

Specify WNPS to enforce the constraint writes no package state, which means that the method does not modify packaged variables.

RNDS

Specify RNDS to enforce the constraint reads no database state, which means that the method does not query database tables.

RNPS

Specify RNPS to enforce the constraint reads no package state, which means that the method does not reference package variables.

TRUST

Specify TRUST to indicate that the restrictions listed in the pragma are not actually to be enforced but are simply trusted to be true.

See Also:

Oracle Database Application Developer's Guide - Fundamentals

---

constructor_spec

Use this clause to create a user-defined constructor, which is a function that returns an initialized instance of a user-defined object type. You can declare multiple constructors for a single object type, as long as the parameters of each constructor differ in number, order, or datatype.

- User-defined constructor functions are always FINAL and INSTANTIABLE, so these keywords are optional.

- The parameter-passing mode of user-defined constructors is always `SELF IN OUT`. Therefore you need not specify this clause unless you wish to do so for clarity.
- `RETURN SELF AS RESULT` specifies that the run-time type of the value returned by the constructor is the same as the run-time type of the `SELF` argument.

See Also:

Oracle Database Application Developer's Guide - Object-Relational Features for more information on and examples of user-defined constructors and "Constructor Example"

---

map_order_function_spec

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

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

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

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

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

See Also:

Oracle Database Application Developer's Guide - Fundamentals for more information about object value comparisons

---

MAP MEMBER

This clause lets you specify 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 induces an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

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

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

Note:

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

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

ORDER MEMBER

This clause lets you specify 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, zero, or positive integer. The negative, positive, or zero indicates that the implicit `SELF` argument is less than, equal to, or greater than the explicit argument.

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

When instances of the same object type definition are compared in an `ORDER BY` clause, the `ORDER` method *`map_order_function_spec`* is invoked.

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

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

create_varray_type

The *`create_varray_type`* lets you create the type as an ordered set of elements, each of which has the same datatype. You must specify a name and a maximum limit of one or more. The array limit must be an integer literal. Oracle Database does not support anonymous varrays.

The type name for the objects contained in the varray must be one of the following:

- A built-in datatype
- A `REF`
- An object type

Restrictions on Varray Types

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 object type column of such a varray type.

See Also:

"Varray Type Example"

---

create_nested_table_type

The *`create_nested_table_type`* lets you create a named nested table of type *`datatype`*.

- If *`datatype`* is an object type, then the nested table type describes a table whose columns match the name and attributes of the object type.
- If *`datatype`* is a scalar type, then the nested table type describes a table with a single, scalar type column called `column_value`.

Restriction on Nested Table Types

You cannot specify `NCLOB` for *`datatype`*. However, you can specify `CLOB` or `BLOB`.

See Also:

"Named Table Type Example" and "Nested Table Type Containing a Varray"

Examples

Object Type Examples

The following 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 the following example, the `data_typ` object type is created with one member function `prod`, which is implemented in the `CREATE TYPE BODY` statement:

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

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

Subtype Example

The following 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)
    );
```

SQLJ Object Type Example

The following examples create a SQLJ object type and subtype. The `address_t` type maps to the Java class `Examples.Address`. The subtype `long_address_t` maps to the Java class `Examples.LongAddress`. The examples specify SQLData as the mechanism used to create the Java instance of these types. Each of the functions in these type specifications has a corresponding implementation in the Java class.

See Also:

Oracle Database Application Developer's Guide - Object-Relational Features for the Java implementation of the functions in these type specifications

---

```
CREATE TYPE address_t AS OBJECT
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
```

```
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME 'create (java.lang.String, java.lang.String, java.lang.String, int) return Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME
        'create (java.lang.String, java.lang.String, java.lang.String, int) return Examples.Address',
    MEMBER FUNCTION to_string RETURN VARCHAR
      EXTERNAL NAME 'tojava.lang.String() return java.lang.String',
    MEMBER FUNCTION strip RETURN SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks () return Examples.Address'
  ) NOT FINAL;

CREATE OR REPLACE TYPE long_address_t
UNDER address_t
EXTERNAL NAME 'Examples.LongAddress' LANGUAGE JAVA
USING SQLData(
    street2_attr VARCHAR(250) EXTERNAL NAME 'street2',
    country_attr VARCHAR (200) EXTERNAL NAME 'country',
    address_code_attr VARCHAR (50) EXTERNAL NAME 'addrCode',
    STATIC FUNCTION create_address RETURN long_address_t
      EXTERNAL NAME 'create() return Examples.LongAddress',
    STATIC FUNCTION  construct (street VARCHAR, city VARCHAR,
        state VARCHAR, country VARCHAR, addrs_cd VARCHAR)
      RETURN long_address_t
      EXTERNAL NAME
        'create(java.lang.String, java.lang.String,
        java.lang.String, java.lang.String, java.lang.String)
          return Examples.LongAddress',
    STATIC FUNCTION construct RETURN long_address_t
      EXTERNAL NAME 'Examples.LongAddress()
        return Examples.LongAddress',
    STATIC FUNCTION create_longaddress (
      street VARCHAR, city VARCHAR, state VARCHAR, country VARCHAR,
      addrs_cd VARCHAR) return long_address_t
      EXTERNAL NAME
        'Examples.LongAddress (java.lang.String, java.lang.String,
         java.lang.String, java.lang.String, java.lang.String)
           return Examples.LongAddress',
    MEMBER FUNCTION get_country RETURN VARCHAR
      EXTERNAL NAME 'country_with_code () return java.lang.String'
  );
```

Type Hierarchy Example

The following 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. For examples, see "Substitutable Table and Column Examples".

Varray Type Example

The following 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);
```

Named Table Type Example

The following example from the sample schema `pm` creates the named table type `textdoc_tab` of object 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;
```

Nested Table Type Containing a Varray

The following 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:

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

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
       );

CREATE TYPE cust_nt_address_typ
   AS TABLE OF cust_address_typ2;
```

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));
```

See Also:

Oracle Database Application Developer's Guide - Fundamentals and *PL/SQL User's Guide and Reference* for more information about constructors

---

Creating a Member Method: Example

The following example invokes method constructor `col.getbar()`. The example assumes the `getbar` method already exists.

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

CREATE TABLE demo_tab2(col demo_typ2);

SELECT col.getbar() FROM demo_tab2;
```

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

Creating a Static Method: Example

The following example changes the definition of the `employee_t` type to associate it with the `construct_emp` function. The example first creates an object type `department_t` and then an object type `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 the following type body statement. The PL/SQL is shown in italics:

```
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));
```