

TEMA 2. BASES DE DATOS ORIENTADAS A OBJETOS

**1. Introducción**

**a. BBDD. Relacionales.**

**b. BBDD O.O.**

i. Nuevos Requerimientos

ii. Estándares de S.A.B.D.O.O

**2. Soporte a la Orientación A Objetos**

**a. Introducción**

**b. Tipos, Abstracción y Clases**

i. Tipos de Creación

ii. Tipos Modificación

iii. Tipos Eliminación

**c. Tipos Colección**

i. Métodos de un tipo de Colección, Varray y Tablas Anidadas

**d. Herencia**

---

# 1. Introducción.

Los sistemas administradores de bbdd nacieron en los años 60.

- La primera generación estaba constituida por los sistemas jerárquicos como IMS y sistemas en red como IDMS.
- La segunda generación estaba constituida por los sistemas relacionales como Oracle, Informix...
- Y la tercera generación está formada por los sistemas administradores de bbdd orientados a objetos (SABDOO).

## 1.1. BD. Relacionales.

Las bbdd relacionales tienen los fundamentos teóricos introducidos por Codd en los años 70 que se basa en una estructura de datos simples llamada relación. Estos sistemas se caracterizan por:

- la persistencia,
- la existencia de un diccionario de datos,
- la abstracción de los datos, la independencia entre programas y datos,
- el manejo de múltiples vistas de usuario, gestión de transacciones concurrentes y
- lenguaje de consulta SQL.

## 1.2. BDOO.

Son la unión entre las características propias de las bbdd y el paradigma de la orientación a objetos. Las bbdd no sólo almacenan información sino también algoritmos, y estas nacen para solucionar los problemas de la persistencia de forma eficiente, sobre todo para sistemas de información geográficos, multimedia, gestión de redes donde los sistemas relacionales han sido incapaces de satisfacer estos requerimientos.

### 1.2.1. Nuevos requerimientos.

- 1) Definición de clases y extensión de los tipos de datos primitivos.
- 2) Objetos complejos. Para poder implementar este tipo de objetos es necesario contar con mecanismos que permitan tener atributos **multivaluados** y referencias a otros objetos.
- 3) Herencia, encapsulamiento y polimorfismo.
- 4) Transacciones de larga duración. Como los objetos y sus atributos pueden ser bastante grandes (ej: video en una bbdd), las transacciones pueden ser bastante largas, por eso se deben considerar los mecanismos de recuperación ante un fallo y el control de consistencia.
- 5) Mecanismos de autorización basados en objetos.

### 1.2.2. Estándares de S.A.B.D.O.O.

Existen dos clases de normas: **de Jure (por ley) de Facto (de hecho)**.

- **Las de Jure** son estándares publicados por organizaciones que se dedican a publicar estándares oficialmente como **ANSI, ISO, IEE...**
- **Las de Facto** son aquellos estándares que realmente están en uso en el mercado, pero no han sido publicados por una organización oficial.

**En bbddoo existen dos estándares:** **el SQL:1999 (tipo Jure)** y **el ODMG (tipo Facto)**.

El primero trata de extender los sistemas administradores de bbdd relacionales para que soporten la OO, y este enfoque es el que han utilizado la mayoría de los fabricantes de bbdd relacionales (**Oracle, Informix...**) y es el que vamos a utilizar.

El segundo confecciona un sistema administrador de bbddoo basado en un modelo de objetos puro que no extiende de los sistemas relacionales.

## 2. Soporte a la orientación a objetos.

### 2.1. Introducción.

Las bdd objeto relacional (como Oracle) son una extensión del modelo relacional que además rompe con ciertas reglas tanto en el modelo relacional como en el paradigma de la orientación a objetos, pero aporta una solución al modelado de la información. La principal diferencia radica en la existencia de atributos multivaluados, que rompe directamente con la primera forma normal. Con este tipo de atributo básicamente se expresa todo lo contrario a la primera forma normal. Se dice que un atributo puede tener más de un valor, pero son del mismo tipo. Para representar atributos multivaluados se usarán las colecciones.

### 2.2. Tipos, abstracción y clases.

#### 2.2.1. Tipos creación.

En Oracle se pueden crear clases a través de los tipos de objetos (tipo=clase).

##### 2.2.1.1.

##### 1) Estructura de un tipo objeto.

Un tipo de objeto tiene dos partes: una especificación y un cuerpo. La parte de la especificación es la interfaz pública, que declara la especificación de atributos y métodos. Y el cuerpo contiene los métodos y es de implementación privada.

##### 2) Sintaxis.

```
Create or replace type nombretipo {is|as} object (atributo_nombre tipo, ... [{member|static} subprograma, ...]);
```

```
Create or replace type body nombretipo {is|as} {member|static} subprograma {is|as}
```

**Member** especifica que el método que se está declarando es tipo miembro.

**Static** significa que el método que se está declarando es de tipo estático, a diferencia de un método miembro se invoca a partir del nombre del tipo y no de un objeto.

**Subprograma** es la especificación de un procedimiento o de una función que consta de un nombre y una lista de parámetros.

Los tipos pueden ser borrados con:

```
drop type nombre_tipo;
```

Ej.)

```
Create or replace type Direccion as object (calle varchar (25), ciudad varchar (20), cpostal number (5));
```

```
Create or replace type Persona as object
```

```
(codigo number,  
nombre varchar (15),  
direc Direccion,  
fecha_nac date);
```

### 3) Persistencia. Mediante manejo tablas (creación, modificación, eliminación): tablas de objeto y tablas relacionales.

En Oracle podemos persistir objetos en columnas de tablas relacionales o en filas de una tabla de objetos. Una **tabla de objetos** es una tabla que almacena un objeto en cada fila, y se accede a los atributos de estos objetos como si se tratase de columnas de tablas.

Ej.)

#### Creación:

```
Create or replace type Punto as object
  (x number,
   y number,
  member procedure moverA (x in number, y in number),
  member procedure moverA (p in Punto),
  member function distanciaA (p in Punto) return number);
```

Para crear cualquiera de las dos tablas, el objeto o tipo al que referimos debe estar creado previamente.

Para crear una tabla relacional es igual una **tabla relacional** normal sólo que alguno de sus campos es del tipo Nombreobjeto:

Ej.)

```
Create table triangulo
  (id varchar (10) primary key,
   A Punto,
   B punto,
   C Punto);
```

Para crear una **tabla de objetos** hay que seguir la siguiente sintaxis:

**Create table nombre of tipo;**

Ej.)

```
Create type Empleado as object
  (rut varchar (10),
   nombre varchar (10),
   cargo varchar (50),
   fecha_ing date,
   sueldo number (9),
   comision number (9),
   anticipo number (9),
  member function sueldo_liquido return number,
  member procedure aumento (sal number);
```

**Create table tabla\_emp of Empleado;**

Al definir tablas de objetos podemos añadir restricciones tipo **not null, check, foreign key**:

Ej.)

```
Create table tabla_emp of Empleado...
  constraint sueldo check (anticipo>0)
  foreign key (deptno) references tbl_emp...
```

Cada instancia de una tabla de objeto posee un identificador único llamado OID, que permite que el objeto almacenado pueda ser referenciado externamente ya sea desde otros objetos o desde columnas de tablas relacionales. El OID lo genera el sistema, y es un número de 16 bits que es único en toda la base de datos.

### Modificación:

Alter table *tabla\_nombre* drop (*nombre\_restriccion*);

El alter table permite **modificar, eliminar** o deshacer restricciones tanto de tablas relacionales como de tablas de objetos (es para lo único que sirve).

### Eliminación:

Drop table *nombre\_tabla*;

Tanto las tablas relacionales como la de objetos se eliminan de esta manera.

## 4) Persistencia. Mediante manejo de objetos:

**Variables de correlación, Alias**  
**(inserción, selección, modificación).**

En general las **variables de correlación** se identifican como los alias de tabla (equivale a ellos). Cuando trabajamos con objetos es obligatorio utilizarlas para acceder a los campos del objeto por separado. Para acceder a dichos campos se utiliza la notación punto.

Ej.)

```
create type tipo_coche as object (  
    marca varchar (25),  
    modelo varchar (25),  
    matricula varchar (9));  
create type tipo_persona as object (  
    nombre varchar (25),  
    coche tipo_coche);  
create table personas of tipo_persona;
```

```
select P.coche.marca from personas P;
```

```
select coche.marca from personas;  
select personas.coche.marca from personas;
```

### Ejemplos de usos de ALIAS

```
create type persona as object (  
    nombre varchar (20));  
create table PTAB1 of persona; *Tabla de objetos de persona. Almacena objetos tipo persona.  
create table PTAB2 (C1 persona); *Es una tabla relacional donde tiene una columna que almacena valores de  
tipo persona.  
create table PTAB3 (C1 ref persona); *Es una tabla relacional que tiene una columna que es una referencia al  
objeto tipo persona.
```

Ej.) select:

```
select P.nombre from PTAB1 P;
```

```
select C1.nombre from PTAB2;  
select P.C1.nombre from PTAB2 P;
```

```
select P.C1.nombre from PTAB3 P;  
select P.nombre from PTAB3 P;
```

**OID** es un identificador (object identifier) para mejorar la búsqueda de información. Cuando se crea una columna cuyo tipo es un tipo objeto, en lugar de almacenar todos los datos del objeto en la propia columna de la tabla, el sistema gestor de bbdd solo almacena un identificador, que será la dirección física de una fila en una tabla auxiliar que el sistema gestor de bbdd usará para almacenar los datos del objeto. Internamente almacena un número de 16 bytes.

**Una referencia** es un puntero a un objeto creado a partir de su OID. Se utiliza para apuntar a un objeto que ya existe y no tener que duplicar la información. Los identificadores únicos asignados por Oracle a los objetos se almacenan en una tabla y permiten que estos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. Los tipos de datos que los soportan se llaman **REF**, y una columna de tipo REF guarda un puntero a una fila de otra tabla, que contiene el OID de dicha fila.

Ej.)

```
create type empleado;  
create type departamento as object (  
    nombre varchar(10),  
    jefe REF empleado);  
create type empleado as object (  
    rut varchar(),  
    nombre varchar(),  
    cargo varchar(),  
    fechaing date,  
    sueldo number(9),  
    comision number(9),  
    anticipo number(9),  
    deptno REF departamento,  
    member function sueldo_liquido return number,  
    member procedure aumento_sueldo (aumento number));  
create table tbl_emp of empleado;  
create table tbl_dept of departamento;
```

#### -Insercion:

Para la inserción utilizaremos el predicado insert into ya sea para la insertar una columna de una tabla relacional, como para insertar en una fila de una tabla de objetos. Para insertar en una tabla relacional:

Ej.)

```
Insert into triangulo values (  
    'triangulo_1',  
    Punto(12,5), //nombre del objeto(valor, valor).  
    Punto(2,25),  
    Punto(19,18)); //Punto(x, y): constructor de ese tipo
```

Para aquellas columnas definidas a partir de un tipo objeto el sistema espera una instancia de ese tipo, y para crearlo utilizaremos el constructor del tipo.

#### Inserción en una tabla de objetos:

Ej.)

```
Insert into tbl_emp values ('11336', 'Miguel', 'Ingeniero', Sysdate, 600000, 1000, 0, null);
```

El orden de los valores debe coincidir con el de los atributos del tipo en el que se basa la tabla de objetos. Si un atributo está definido en base a un tipo objeto a la hora de hacer el insert, será necesario usar su constructor. Para las columnas que sean de tipo REF necesitamos utilizar una subconsulta (dentro del insert) para obtener sus valores.

Ej.)

```
Insert into tbl_emp values ('113307', 'Juan', 'Administrativo', Sysdate, 2000,1000, (select REF(d) from tbl_dept d where d.nombre='RRHH'));
```

```
Insert into tbl_dept values('Ventas',(select REF(e) from tbl_emp e where e.rut='11336'));
```

### -Selección:

Para realizar consultas se usa el predicado select sobre las tablas de objetos como si fueran tablas relacionales, donde los atributos del objeto son las columnas de la tabla. En las consultas se pueden invocar a funciones miembros de un objeto, y utilizarlas como si fueran columnas.

Ej.)

```
Select * from tbl_emp e where e.sueldo>100;
```

```
Select rut, nombre, e.sueldo_liquido() from tbl_emp e where e.sueldo_liquido<100;
```

```
Select * from tbl_emp;
```

\*En el deptno que es una referencia nos va a salir su OID, para evitar que salga y ver los datos usar **DEREF**:

```
Select rut, nombre, DEREF(deptno) from tbl_emp;
```

```
Select rut, nombre, e.deptno.nombre from tbl_emp e; // solo valido en SQL no en PL
```

```
Select rut, nombre, DEREF(deptno).nombre from tbl_emp;
```

### -Modificación:

Ej.)

```
update tbl_emp emp set emp.sueldo=emp.sueldo*1.15 where emp.sueldoLiquido()<100;
```

También se puede reemplazar el valor de todos los atributos del objeto con los de otro:

```
update tbl_emp emp set emp=Empleado('131344', 'Miguel R.', 'Ingeniero', Sysdate, 800000, 2000, 0, null) where emp.rut='131344';
```

Actualizar el atributo Deptno que es una referencia a una instancia de departamento:

```
update tbl_emp emp set emp.deptno=(select REF(d) from tbl_dept d where d.nombre='INFORMATICA') where emp.rut='131344';
```

### -Delete:

Eliminación de objetos.

Ej.)

```
delete from tbl_emp emp where emp.sueldoLiquido()<100;
```

Elimina todos los empleados cuyo sueldo líquido sea menor que 100.

## 5) Operador value.

Ej.)

```
create type tipo_coche as object (  
    marca varchar (25),  
    modelo varchar (25),  
    matricula varchar (9));  
create type tipo_persona as object (  
    nombre varchar (25),  
    coche tipo_coche);  
create table persona of tipo_persona;
```

Para obtener el objeto almacenado en una fila, y no sólo el valor de los campos de dicho objeto se necesita utilizar el operador **value**.

Ej.)

```
Select * from personas;
```

```
⌘ramon ramirez tipo_coche ('citroen', '20', 'M-9978');
```

Sólo devuelve el valor de los campos del objeto de la clase tipo\_persona.

Ej.)

```
Select value(p) from personas p;
```

```
⌘tipo_persona('ramon ramirez', tipo_coche ('citroen', '20', 'M-9978'));
```

Devuelve el objeto entero de la clase tipo\_persona.

Ej.)

```
declare
```

```
    v_persona tipo_persona;
```

```
begin
```

```
    select value(p) into v_persona from persona p where nombre like '%Ram%';  
    dbms_output.put_line(v_persona.nombre);  
    dbms_output.put_line(v_persona.coche.marca);  
    dbms_output.put_line(v_persona.marca.modelo);
```

```
end;
```

```
/
```

Ej2.)

```
declare
```

```
    v_persona tipo_persona;
```

```
begin
```

```
    v_persona:=tipo_persona('Roberto',tipo_coche('seat','600','b-8888'));  
    insert into personas values(v-persona);
```

```
end;
```

```
/
```



### 2.2.1.2.

#### 1) Atributos.

Todo tipo de objeto debe poseer un atributo como mínimo y máximo 1000. El atributo constará del nombre(único por objeto) y un tipo de dato (cualquiera de los tipos primitivos a excepción del LONG y el LONG ROW).

Entre los tipos primitivos existe uno denominado REF que permite almacenar una referencia a un tipo de objeto. Gracias a este tipo podemos definir un tipo de objeto recursivo, que es aquel en el que uno o más de sus atributos tienen como tipo de dato una referencia a sí mismo.

Ej.)

```
create or replace type lista as object(  
    info number,  
    next REF lista);
```

Un tipo no puede tener un atributo del mismo tipo que él.

```
create or replace type lista as object(  
    info number,  
    next lista);--- ERROR!!! PLS_00318 it's a non_REF
```

Si comparamos la definición de atributos con la definición de columnas de una tabla, son muy similares, aunque existen diferencias:

#### **DIFERENCIAS ENTRE COLUMNAS DE TABLA Y DEF DE ATRIBUTOS**

1. En los tipos no se pueden poner valores iniciales ni se pueden poner restricciones del tipo primary key.

##### **Ej.) CREAMOS EL TIPO EMPLEADO**

```
create or replace type empleado as object(  
    rut varchar(10),  
    nombre varchar(30),  
    fecha_ingreso date,  
    salario number);
```

##### **CREAMOS LA TABLA EMPLEADO**

```
create table empleado(  
    rut varchar(10) primary key,  
    nombre varchar(20),  
    fecha date default sysdate,  
    sueldo numero constraint ck_sueldo check(sueldo>95000));
```

2. El tipo de dato de un atributo debe existir previamente. Este problema se puede producir cuando existen dos tipos mutuamente dependientes (uno depende de otro a través de una referencia). Para solucionar el problema existe una sentencia (definición de tipos adelantada), con la que generamos un tipo objeto incompleto:

```
create type nombre_type;
```

```
Ej.)  
create type empleado;  
create type departamento as object (  
    nombre varcgar (10),  
    jefe REF empleado);  
create type empleado as object (  
    rut varchar (),  
    nombre varchar (),  
    deptno REF departamento);
```

3. No se puede definir un atributo a partir de un tipo incompleto, pero si una referencia.

```
create type departamento as object (  
    nombre varcgar (10),  
    jefe empleado); --error SOLO SE ADMITEN REFERENCIAS A TIPOS INCOMPLETOS
```

2) **OID** es un identificador (object identifier) para mejorar la búsqueda de información. Cuando se crea una columna cuyo tipo es un tipo objeto, en lugar de almacenar todos los datos del objeto en la propia columna de la tabla, el sistema gestor de bbdd solo almacena un identificador, que será la dirección física de una fila en una tabla auxiliar que el sistema gestor de bbdd usará para almacenar los datos del objeto. Internamente almacena un número de 16 bytes.

```
Select * from tbl_emp;
```

\*En el deptno que es una referencia nos va a salir su OID, para evitar que salga y ver los datos usar Deref:

```
Select rut, nombre, Deref(deptno) from tbl_emp;
```

```
Select rut, nombre, e.deptno.nombre from tbl_emp e; * valida solo en sql
```

```
Select rut, nombre, Deref(deptno).nombre from tbl_emp;
```

\*Para acceder dentro de ref deptno a los datos, se puede hacer de las dos maneras anteriores (la primera vale sólo para SQL).

### **OID (object identifier)**

**Para ver el OID tenemos las columnas object\_id, object\_value**

Ej.)

```
create type coche as object (  
    marca varchar(20),  
    modelo varchar(200));  
create table vehiculos of coche;
```

```
select object_id,object_value from vehiculos;
```

**object\_id:** devuelve el código.

**object\_value:** devuelve los valores del objeto.

### 3) Tipos referencia (REF).

**Referencia** es un puntero a un objeto creado a partir de su OID. Se utiliza para apuntar a un objeto que ya existe y no tener que duplicar la información. Los identificadores únicos asignados por Oracle a los objetos se almacenan en una tabla y permiten que estos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. Los tipos de datos que los soportan se llaman REF, y una columna de tipo REF guarda un puntero a una fila de otra tabla, que contiene el OID de dicha fila.

Ej.)

```
create type empleado_t as object(  
    nombre varchar(30),  
    jefe ref empleado_t);  
create table empleado of empleado_t;  
insert into empleado values('arroyo',(select ref(e) from empleado e where e.nombre='gil'));  
select nombre,deref(p.jefe) from empleado;
```

```
Insert into tbl_emp values ('113307', 'Juan', 'Administrativo', Sysdate, 2000,1000, (select REF(d) from tbl_dept  
d where d.nombre='RRHH'));  
Insert into tbl_dept values('Ventas',(select REF(e) from tbl_emp e where e.rut='11336'));
```

También se puede reemplazar el valor de todos los atributos del objeto con los de otro:

```
update tbl_emp emp set emp=Empleado('131344', 'Miguel R.', 'Ingeniero', Sysdate, 800000, 2000, 0, null)  
where emp.rut='131344';
```

Actualizar el atributo Deptno que es una referencia a una instancia de departamento:

```
update tbl_emp emp set emp.deptno=(select REF(d) from tbl_dept d where d.nombre='INFORMATICA') where  
emp.rut='131344';
```

*Dar los ejemplos de inserción, modificación y borrado con referencias aquí, están en verde*

### 2.2.1.3.

#### 1) Métodos.

Son funciones o procedimientos almacenados que se asocian a un tipo específico y no pueden ser invocados independientemente. Pueden ser de tres tipos:

##### -Constructor.

Todos los tipos objeto definidos por el usuario tienen un método constructor generado por el sistema, que es una función con el mismo nombre que el tipo. Los parámetros formales de esta función coinciden exactamente con los atributos del tipo, es decir, en el mismo orden, con el mismo nombre y con el mismo tipo de datos. El valor de retorno del constructor es un nuevo objeto del mismo tipo, al que pertenece el constructor, donde los valores de los atributos fueron inicializados con los valores pasados por parámetro al llamar al constructor.

Ej.) **Constructor predefinido Oracle**

```
create or replace type tipo_coche as object (  
    marca varchar(25),  
    modelo varchar(25),  
    matricula varchar(9));  
create or replace type tipo_persona as object (  
    nombre varchar(25),  
    coche tipo_coche);  
create table personas of tipo_persona;  
insert into personas values('Ramon', tipo_coche('Citroen', '2cv', 'M999'));
```

Ej.) **Constructor no predefinido**

```
create or replace type Rectangulo as object (  
    base number,  
    altura number,  
    area number,  
    constructor function Rectangulo (base number, altura number) return self as result);  
  
create or replace type body rectangulo as  
    constructor function Rectangulo (base number, altura number) return self as result as  
        begin  
            self.base:=base;  
            self.altura:=altura;  
            self.area:=base*altura;  
            return;  
        end;  
end;
```

### -Miembros.

Hay que especificar la palabra member y son invocados a partir de un objeto. Son definidos por el usuario en el momento de especificar el tipo, pudiendo ser una función o un procedimiento.

Ej.)

```
create or replace type Punto as object (  
    x number,  
    y number,  
    member procedure moverA (x in number, y in number),  
    member function distanciaA (P in Punto) return number);  
create or replace type body Punto as  
    member procedure moverA (x in number, y in number) is  
    begin  
        self.x:=x;  
        self.y:=y;  
    end;  
    member function distanciaA (P in Punto) return number is  
    begin  
        return SQRT(power((x-P.x),2)+power(y-P.y),2));  
    end;  
end;
```

El argumento **self** está implícito al declarar el método, este parámetro nos permite acceder a los atributos o métodos del objeto y su uso sólo es obligatorio cuando un método cuenta con un parámetro con el mismo nombre que un atributo. El modo por defecto de este parámetro es **in** para el caso de las funciones miembro, e **inout** para el caso de los procedimientos miembro, esto significa que no podemos actualizar los atributos del objeto en una función, pero sí en un procedimiento.

Las funciones miembros pueden ser invocadas desde sentencias SQL siempre que el modo de todos sus parámetros sea **in**.

Los procedimientos miembro no pueden ser invocados desde sentencias sql, solo a través de objetos transitorios desde el lenguaje de programación.

Ej.)

```
create or replace type Direccion as object (  
    calle varchar(25),  
    ciudad varchar(10),  
    codigo_post number(5),  
    member procedure set_calle (C varchar),  
    member function get_calle return varchar);  
create or replace type body Direccion as  
    member procedure set_calle (C varchar) is  
        begin  
            calle:=C;  
        end;  
    member function get_calle return varchar is  
        begin  
            return calle;  
        end;  
end;  
/  
declare  
    dir Direccion:=Direccion(null, null, null);  
begin  
    dir.set_calle('Gandia');  
    dbms_output.put_line(dir.get_calle);  
end;  
/
```

### **-Estáticos.**

Un método estático, al igual que un método member, puede ser un procedimiento o una función, pero a diferencia de este no podemos declarar el parámetro self ni implícita, ni explícitamente. Por ello en un método estático no podemos acceder a los atributos del tipo. Son independientes de las instancias del objeto, su uso más común es para crear funciones que hagan las veces de constructor.

Ej.)

```
create or replace type Punto as object (  
    X number,  
    Y number,  
    member procedure moverA (x in number, y in number),  
    member function distancia A (p in Punto) return number,  
    static function nuevoPuntoConstructor (p in Punto) return Punto);  
  
Invocar con: Punto.nuevoPuntoConstructor(P);
```

### 2.2.2 Tipos modificación.

El predicado ALTER TYPE permite recompilar la especificación y/o el cuerpo de un tipo objeto, también permite agregarle nuevos métodos.

Con alter type no podemos modificar o eliminar los atributos o métodos existentes, tampoco agregar atributos.

```
Alter type nombre {compile[{specification | body}] | replace as object (
    Atributo tipo,
    Atributo2 tipo2,
    [{member | static} subpr]}
```

Ej.)

```
alter type Empleado compile body; //compila solo el body
```

```
alter type Empleado compile; //compila todo
```

```
// agregando un nuevo método anticipo
```

```
alter type Empleado replace as object (
    rut varchar(10),
    nombre varchar(15),
    cargo varchar(9),
    fecha_ing date,
    sueldo number(9),
    comision number (9),
    anticipo number(9),
    member function sueldo_liq return number,
    member procedure comision (aumneto number),
    member procedure setAnticipo (anticipo number));
```

```
create or replace type body Empleado is
    member function...
```

```
-
```

```
-
```

```
end;
```

```
member procedure...
```

```
-
```

```
-
```

```
end;
```

```
member procedure...
```

```
-
```

```
-
```

```
end;
```

```
end;
```

### 2.2.3. Tipos eliminación.

**Drop type *nombre* [force];**

\*fuerza el borrado del tipo en caso de que existan referencias.

**Drop type body *nombre*;**

\*Borra sólo el cuerpo.



## 2.3. Tipos colección.

Un tipo colección es un tipo de dato definido por el usuario que nos permite almacenar un número indefinido de elementos, todos de un mismo tipo. Las colecciones se utilizan para representar atributos multivaluados. Estos atributos rompen la primera forma normal y dicen que un atributo puede tener más de un valor del mismo tipo.

Id	Nombre	Apellidos	Hijos (estos son atributos multivaluados)
1	Pepe	Lopez	Luis, Ana, María
2	Juan	Gomez	Carlos, Pedro

En general al igual que un tipo objeto, un tipo colección posee un método constructor definido por el sistema cuyo nombre es el mismo que el de la colección, y los parámetros corresponden a los elementos que contendrá separados por coma, si no se pasan parámetros se crea una colección vacía, que es diferente de una colección nula.

En Oracle existen dos tipos de colecciones:

- Varrays
- Nested table (tablas anidadas).

### 2.3.1. Métodos de un tipo colección.

Exists(n)		Retorna true si el elemento n existe.
Count		Cuenta el número de elementos de la colección.
Limit		En varrays sólo, número máximo de elementos que puede contener.
First, Last		Retorna el primer y último índice de una colección.
Prior(n), Next(n)		Retorna el índice antecesor o sucesor al elemento ubicado en la posición n.
Agrandan el tamaño de una colección	Extend	Añade un elemento null al final de la colección.
	Extend(n)	Añade n elementos null al final de la colección.
	Extend(n,i)	Añade n copias del elemento con posición i al final de la colección.
Trim		Elimina el último elemento.
Trim(n)		Elimina n elementos desde el final.
Eliminan elementos	Delete	Elimina todos los elementos de la colección (para Varrays).
	Delete(n)	Elimina el elemento de la posición n de la tabla anidada (para nested table).
	Delete(m,n)	Elimina los elementos desde m hasta n de la tabla anidada (para nested table).

### 2.3.2. Varray.

Es una colección de elementos del mismo tipo donde cada elemento está en una posición determinada con un índice. Los varrays son de longitud variable, pero hay que indicar el máximo número de elementos que puede contener.

Un tipo varray se puede utilizar para varios fines:

- Definir un tipo de dato de una columna de una tabla relacional.
- Definir un tipo de dato de un atributo de un tipo objeto.
- Definir una variable PLSQL, un parámetro o el tipo de retorno de una función.

Cuando se declara un varray no se reserva ningún espacio, se almacena junto con el resto de las columnas de la tabla, salvo que sea demasiado largo (>4KB), que se almacena en una tabla llamada blob.

**Create type *nombre* as VARRAY (size) of tipo [not null];**

Ej.)

*Create or replace type precios as varray (10) of number(12);*

\*Dentro de tipo no se pueden poner: binary\_integer, boolean, long, string, table, varray...

Ej.) Varray de objetos:

*create or replace type arr\_puntos as varray(100) of punto;*

*Polilinea:= arr\_puntos(Punto(3, 4), Punto(5, 1)); //constructor*

*Polilinea.Extend; //agrega un elemento null al final*

*Polilinea(3):=Punto (8, 5); //Agrega el elemento punto (8,5) en la posición 3.*

*Polilinea.trim; //elimina el último elemento.*

Ej.) 2.

*create type Lista\_tel\_t as varray(10) of varchar(20);*

*create type Cliente\_t as object (*

*c\_num number,*

*c\_nom varchar (20),*

*direc direccion\_t,*

*Lista\_tel Lista\_tel\_t);*

Ej.) 3.

*create or replace type tipo\_pers as object (*

*nombre varchar (25),*

*dni varchar (10));*

*create or replace type tipo\_personas as varray(15) of tipo\_pers;*

*create or replace type tipo\_empresa as object (*


*nombre varchar(15),*

*nif varchar(15),*

*empleados tipo\_personas);*

*create table empresas of tipo\_empresa;*

*insert into empresas values('ACME', '77654K', tipo\_personas(tipo\_pers('pepe perez', '123456789A')));*

  
constructor de tipo colección

Existe una vista del diccionario de datos donde se pueden consultar las tablas que tienen columnas Varrays.

- **USER\_VARRAYS**
- **DESC nombre\_varray**

### 2.3.3. Tablas anidadas.

Una tabla anidada es un conjunto de elementos sin orden particular todos del mismo tipo, y sirven para no limitar el número de valores dentro del atributo multivaluado, sólo es necesario especificar el tipo.

Esta posee una única columna cuyo tipo puede ser un tipo primitivo o un tipo objeto. Si es un tipo objeto, la tabla puede ser vista como una tabla de múltiples columnas, con una columna para cada atributo de tipo objeto.

Oracle almacena una tabla anidada sin un orden particular, pero cuando se recupera una tabla anidada desde la base de datos, y se almacena en una variable PL a cada fila se le asigna una posición, empezando por el uno, y esto permite acceder a la tabla anidada como si fuera un varray.

Las diferencias básicas entre un varray y una tabla anidada son:

1. Un varray está limitado a un número máximo de elementos, pero en las tablas anidadas no.
2. En un varray no podemos eliminar directamente elementos intermedios, pero en una tabla anidada sí.
3. Oracle almacena los datos de un varray en la misma tabla (los almacena en línea), y en una tabla anidada se almacenan fuera de línea (en una tabla generada por el sistema).
4. Cuando almacenamos en la base de datos un varray, este retiene su orden y subíndices, pero las tablas anidadas no.
5. Desde SQL no podemos insertar, modificar o eliminar elementos de un varray, pero sí lo podemos hacer de una tabla anidada.
6. Podemos crear índices en tablas anidadas, pero no en un varray

SINTAXIS:

```
CREATE TYPE nombre_tipo_tabla_anidada IS TABLE OF tipo [NOT NULL];
```

```
CREATE TABLE nombre_tabla_anidada [OF TIPO_OBJ] *  
  (campo1 tipo,  
   campo2 nombre_tipo_tabla_anidada)  
  NESTED TABLE campo2 STORE AS nombre_tabla;
```

\*La tabla puede ser de objetos o relacional (sin poner of tipo\_obj).

La cláusula **nested table** identifica el nombre de la columna que contendrá la tabla anidada.  
La cláusula **store as** especifica el nombre de la tabla anidada en la cual se van a almacenar los datos.

Para información de las tablas anidadas:

```
desc nombre_tipo_tabla_anidada;  
desc nombre_tabla_anidada;
```

Se puede hacer un select a la vista **user\_nested\_tables** (da información sobre las tablas anidadas).

```
Select * from user_nested_tables;
```

La tabla **dba\_objects** sirve para consultar objetos de la bbdd, y **dba\_segments** consulta la estructura de almacenamiento que usa Oracle para almacenar un objeto.

Ej.)

```
Select object_name from dba_objects where owner='daw2';
```

```
Select segment_name , segment_type from dba_segments where segment_name like '%hijos%';
```

Ej.) Crear una tabla anidada para almacenar objetos de tipo dirección (suponiendo que ya lo tengamos con calle, ciudad y cp):

```
Créate or replace type Dirección as object( calle varchar(25), ciudad varchar(25), cpostal number(15));
```

```
/
```

```
Create type t_anidada as table of Dirección;
```

```
/
```

```
Create table ej_t_anidada (
```

```
    Id number (2),
```

```
    Apellidos varchar(25),
```

```
    Direc t_anidada)
```

```
    Nested table Direc store as direc_anidada;
```

```
/
```

\*direc\_anidada: es la tabla anidada en sí. No se puede operar sobre ella directamente (select, insert...), sino que hay que acceder a ella mediante la tabla en sí (ej\_t\_anidada)

Ej.) Insertar dos filas en la tabla anterior:

```
Insert into ej_t_anidada values(1, 'ramos',  
    t_anidada(  
        dirección('sol', 'guadalajara', 29111),  
        dirección('via', 'madrid', 28007),  
        dirección('montaña', 'caceres', 35457)));
```

```
insert into ej_t_anidada values(2, 'martin',  
    t_anidada(  
        dirección('huesca', 'albacete', 32114),  
        dirección('nuño', 'toledo', 7777)));
```

```
select * from ej_t_anidada;
```

**Ej.) Modificar:**

```
Update ej_t_anidada set
  Direc=t_anidada(
    dirección('dunas', 'madrid', 28444),
    dirección('luna', 'valencia', 8217),
    dirección('cielo', 'burgos', 2722)) where id=1;
```

Modificar todas las direcciones de la tabla anidada, solo es posible hacerlo así, usando el constructor.

**·Operador TABLE-----:**

Se usa para acceder a la fila y columna en concreto que nos interesa dentro de la tabla anidada.

La consulta estándar recupera los datos anidados, pero si quieres sacar una columna en concreto, por ejemplo la calle, hay que usar el operador TABLE, que permite descomponerlos.

**Ej) consulta estándar:**

```
Select E.direc from ej_t_anidada E where W.id = 1;
```

Salida:

```
DIREC(calle,cuidad,cp) t_anidada(Direccion('calle sol','guadalajara','29888')
                                   Direccion('calle luna','toledo','67888')
                                   Direccion('calle estrella','murcia','90877'));
```

**Ej.) Sacar todas las direcciones de la persona id=1:**

```
Select e.direc from ej_t_anidada e where e.id=1;
```

**CONSULTA USANDO TABLE:**

```
SELECT ..... from tabla1 x , TABLE(x.campo) alias WHERE.....
```

**Ej.) Solo sacar las calles de la fila con id=1 cuya ciudad sea Guadalajara:**

```
Select dir.calle from ej_t_anidada e, table(e.Direc) dir where e.id=1 and dir.ciudad='guadalajara';
```

**dir:** alias de la tabla que hace referencia a Direc.

**Direc:** el nombre del campo que contiene t\_anidada.

**Ej.) Crear un procedimiento PL que recibe un id y visualiza las calles:**

```
Create or replace procedure ver_calles (ident number) as
  Cursor is select dir.calle form ej_t_anidada e, table(e.direc) dir where e.id=ident;
Begin
  For i in c1 loop
    Dbms_output.put_line(i.calle);
  End loop;
End;
```

**Para insertar usando TABLE**, un solo campo en la tabla anidada (se inserta al final):

**Insert into TABLE(subconsulta) values (tuplas a insertar);**

Ej.)

*Insert into table(select e.direc from ej\_t\_anidada e where id=1) values(dirección('c/manantiales,15',  
'guadalajara', 72820);*

\*El operador **table()** se utiliza siempre para trabajar con la tabla anidada, para el resto de las tablas se trabaja normal.

**Modificación usando TABLE:**

**Update table(subconsulta) set...**

Ej.) Modificar la primera dirección del id=1.

*Update TABLE(select e.direc from ej\_t\_anidada e where id=1) primera set value(primer)=dirección('c/pintor  
11', 'toledo', 4599) where value(primer)=dirección('c/los naranjos', 'guadalajara', 7877);*

*Primera:* recoge los datos devueltos por el select.

*Value:* para obtener el objeto almacenado y no sólo el valor de los campos de dicho objeto.

Ej) Para Modificar todas las direcciones de la tabla anidada, solo es posible hacerlo así, usando el constructor.

```
Update ej_t_anidada set
  direc=t_anidada(
    dirección('dunas', 'madrid', 28444),
    dirección('luna', 'valencia', 8217),
    dirección('cielo', 'burgos', 2722)) where id=1;
```

**Borrar:**

Ej.) Borrar la dirección insertada anteriormente:

*Delete from table(select e.direc from ej\_t\_anidada e where id=1) primera where  
value(primer)=dirección('c/los manantiales', 'Guadalajara', 72800);*

## 2.4. Herencia.

Una de las ventajas de la PPOO es la herencia, gracias a la cual se pueden crear subclases más específicas que hereden los atributos y métodos de la superclase. En BBDDOO es posible hacer algo parecido con los tipos de objeto, se pueden crear objetos a partir de otros ya existentes, lo que implica que un subtipo obtendrá todo el comportamiento (atributos y métodos) del supertipo. Los subtipos pueden definir sus propios atributos y métodos y redefinir los métodos que heredan (polimorfismo).

- **Not final:** se pueden definir más tipos a partir de él.
- **Final:** los subtipos no pueden redefinirlo.
- **Under:** especifica a que supertipo pertenece.
- **Overriding:** redefinir.

Ej.)

```
Create or replace type tipo_persona as object (  
    Id number,  
    Nombre varchar(10),  
    Teléfonos Listin)Not final;  
  
/  
Create or replace type tipo_estudiante under tipo_persona (  
    Facultad varchar(10),  
    Nota_media number)Not final;  
  
/
```

Ej.)

```
Create or replace type tipo_persona as object (  
    Dni varchar(10),  
    Nombre varchar(25),  
    Fecha_nac date,  
    Member function edad return number,  
    Final member function get_dni return varchar,  
    Final member function get_nombre return varchar,  
    Member procedure ver_datos)Not final;  
  
/  
Create or replace type body tipo_persona as  
    Member function edad return number as  
    Ed number;  
    Begin  
        Ed:=to_char(sysdate, 'yyyy')-to_char(fecha_nac, 'yyyy');  
        Return ed;  
    End;  
    Final member function get_dni return varchar  
    Begin  
        Return dni;  
    End;  
    Member procedure ver_datos as  
    Begin  
        Dbms_output.put_line(dni||' * '||nombre||' * '||edad());  
    End;  
    Final member function get_nombre return varchar as  
    Begin  
        Return nombre;  
    End;  
End;  
/
```

```

Create or replace type tipo_alumno under tipo_persona(
    Curso varchar(10),
    Nota_final number,
    Member function nota return number,
    Overriding member procedure ver_datos);
Create or replace type body tipo_alumno as
    Member function nota return number as
    Begin
        Return nota_final;
    End;
Overriding member procedure ver_datos as
    Begin
        Dbms_output.put_line(curso||' * '|nota_final);
    End;
End;
/

```



```

Create or replace type tipo_persona as object (
    Id number,
    Nombre varchar(10),
    Teléfonos Listin)Not final;
/

Create or replace type tipo_estudiante under tipo_persona (
    Facultad varchar(10),
    Nota_media number)Not final;
/

Create or replace type tipo_persona as object (
    Dni varchar(10),
    Nombre varchar(25),
    Fecha_nac date,
    Member function edad return number,
    Final member function get_dni return varchar,
    Final member function get_nombre return varchar,
    Member procedure ver_datos)Not final;
/

Create or replace type body tipo_persona as
    Member function edad return number as
    Ed number;
    Begin
        Ed:=to_char(sysdate, 'yyyy')-to_char(fecha_nac, 'yyyy');
        Return ed;
    End;
    Final member function get_dni return varchar
    Begin
        Return dni;
    End;
    Member procedure ver_datos as
    Begin
        Dbms_output.put_line(dni||' * '||nombre||' * '||edad());
    End;
    Final member function get_nombre return varchar as
    Begin
        Return nombre;
    End;
End;
/

Create or replace type tipo_alumno under tipo_persona(
    Curso varchar(10),
    Nota_final number,
    Member function nota return number,
    Overriding member procedure ver_datos);
Create or replace type body tipo_alumno as
    Member function nota return number as
    Begin
        Return nota_final;
    End;
    Overriding member procedure ver_datos as
    Begin
        Dbms_output.put_line(curso||' * '||nota_final);
    End;
End;
/

```

El siguiente ejemplo muestra el uso de los tipos definidos anteriormente.

Al definir el objeto se inicializan todos los atributos, ya que no se ha definido constructor para inicializar el objeto.

*Declare*

```
A1 tipo_alumno:= tipo_alumno(null,null,null,null,null);
A2 tipo_alumno:= tipo_alumno('643643a', 'pedro','12/12/1996',7);
nom A1.nombre%type;
dni A1.dni%type;
notaf A1.nota_final%type;
```

*BEGIN*

```
A1.nota_final:=8;
A1.curso:='primero';
A1.nombre:='juan';
A1.fec_nac:='20/01/1997';
A1.ver_datos;
nom:=A2.get_nombre();
dni:=A2.get_dni();
notaf:= A2.nota();
A2.ver_datos;
DBMS_OUTPUT.PUT_LINE(A1.edad());
DBMS_OUTPUT.PUT_LINE(A2.edad());
```

*END;*

**Declare**

```
A1 tipo_alumno:= tipo_alumno(null,null,null,null,null) ;
A2 tipo_alumno:= tipo_alumno('643643a', 'pedro','12/12/1996',7) ;
nom A1.nombre%type;
dni A1.dni%type;
notaf A1.nota_final%type;
```

**BEGIN**

```
A1.nota_final:=8;
A1.curso:='primero';
A1.nombre:='juan';
A1.fec_nac:='20/01/1997';
A1.ver_datos;
nom:=A2.get_nombre();
dni:=A2.get_dni();
notaf:= A2.nota();
A2.ver_datos;
DBMS_OUTPUT.PUT_LINE(A1.edad());
DBMS_OUTPUT.PUT_LINE(A2.edad());
```

**END;**

A continuación se crea una tabla de tipo\_alumno con el dni como clave primaria, se insertan filas y se realizan consultas.

```
Create table Talumnos of tipo_alumno(dni primary key);
insert into Talumnos values ('8988877A','pedro','12/02/1996','segundo',7);
insert into Talumnos values ('654645456A','juan','23/06/1997','tercero',8);
select * from Talumnos;
select dni,nombre,curso, nota_final from Talumnos;
select P.get_dni(), P.get_nombre(), P.edad(), P.nota() from Talumnos P;
```

```
Create table Talumnos of tipo_alumno(dni primary key);
insert into Talumnos values ('8988877A','pedro','12/02/1996','segundo',7);
insert into Talumnos values ('654645456A','juan','23/06/1997','tercero',8);
select * from Talumnos;
select dni,nombre,curso, nota_final from Talumnos;
select P.get_dni(), P.get_nombre(), P.edad(), P.nota() from Talumnos P;
```