4.1. INTRODUCCIÓN

Las bases de datos constituyen una de las piezas fundamentales de muchos sistemas de información, muchas de ellas, sobre todo las más tradicionales, son difíciles de utilizar cuando las aplicaciones que acceden a los datos utilizan lenguajes de programación orientado a objetos como C++ o Java. Este fue uno de los motivos de la creación de las bases de datos orientadas a objetos, además de dar solución al surgimiento de aplicaciones más sofisticadas que necesitan tipos de datos y operaciones más complejas.

Los fabricantes de SGBD relacionales han ido incorporando en las nuevas versiones muchas de las propuestas para las bases de datos orientadas a objetos, un ejemplo son Informix, Oracle o PostgreSQL. Esto ha dado lugar al modelo relacional extendido y a los sistemas que lo implementan que son los llamados **sistemas Objeto-Relacionales**.

4.2. BASES DE DATOS OBJETO-RELACIONALES

Las **Bases de Datos Objeto-Relacionales (BDOR)** son una extensión de las bases de datos relacionales tradicionales a las que se les ha añadido conceptos del modelo orientado a objetos, por tanto, un **Sistema de Gestión de Base de Datos Objeto-Relacional (SGBDOR)** contiene características del modelo relacional y del orientado a objetos; es decir, es un sistema relacional que permite almacenar objetos en las tablas.

4.2.1. Características

Los modelos de datos relacionales orientados a objetos extienden el modelo de datos relacional proporcionando un sistema de tipos más rico e incluyendo tipos de datos complejos y la programación orientada a objetos. Los lenguajes de consulta relacionales como SQL también necesitan ser extendidos para trabajar con estos nuevos tipos de datos. Las extensiones orientadas a objetos que comúnmente se encuentran en las bases de datos relacionales orientadas a objetos son: objetos de datos de gran tamaño, tipos de datos estructurados/abstractos, tipos de datos definidos por el usuario, tablas en tablas, secuencias, conjuntos y arrays, procedimientos almacenados, etc. Las características orientadas a objetos se definieron en el estándar SQL:1999.

En definitiva, las características más importantes de los SGBDOR son las siguientes:

- Soporte de tipos de datos básicos y complejos. El usuario puede crear sus propios tipos de datos.
- Soporte para crear métodos para esos tipos de datos. Se pueden crear funciones miembro usando tipos de datos definidos por el usuario.
- Gestión de tipos de datos complejos con un esfuerzo mínimo.
- Herencia.
- Se pueden almacenar múltiples valores en una columna de una misma fila.
- Relaciones (tablas) anidadas.
- Compatibilidad con las bases de datos relacionales tradicionales. Es decir, se pueden pasar las aplicaciones sobre bases de datos relacionales al nuevo modelo sin tener que rescribirlas.

- El **inconveniente** de las BDOR es que aumenta la complejidad del sistema, esto ocasiona un aumento del coste asociado.

En este apartado estudiaremos la orientación a objetos que proporciona Oracle

4.2.2. Tipos de objetos

Para crear tipos de objetos utilizamos la orden **CREATE TYPE** (OR REPLACE reemplaza el tipo si ya existe). El siguiente ejemplo crea dos objetos, un objeto que representa una dirección formada por tres atributos: calle, ciudad y código postal, cada uno de los cuales con su tipo de dato; y el siguiente representa una persona con los atributos código, nombre, dirección y fecha de nacimiento:

```
CREATE OR REPLACE TYPE DIRECCION AS OBJECT
(
  CALLE  VARCHAR2(25),
  CIUDAD VARCHAR2(20),
  CODIGO_POST NUMBER(5)
);
/
CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
  CODIGO NUMBER,
  NOMBRE VARCHAR2(35),
  DIREC  DIRECCION,
  FECHA_NAC DATE
);
/
```

Oracle responderá con el mensaje: *Tipo creado* para cada tipo creado (desde la línea de comandos de SQL). Una vez creados podemos usarlos para declarar e inicializar objetos como si se tratase de cualquier otro tipo predefinido, hemos de tener en cuenta que al declarar el objeto dentro de un bloque PL/SQL hemos de inicializarlo. El siguiente ejemplo muestra la declaración y uso de los tipos creados anteriormente:

```
DECLARE
  DIR DIRECCION := DIRECCION(NULL,NULL,NULL);
  P PERSONA     := PERSONA(NULL,NULL,NULL,NULL);
  DIR2 DIRECCION;   -- SE INICIA CON NEW
  P2 PERSONA;       -- SE INICIA CON NEW
BEGIN
  DIR.CALLE := 'La Mina, 3';
  DIR.CIUDAD := 'Guadalajara';
  DIR.CODIGO_POST := 19001;
  --
  P.CODIGO := 1;
  P.NOMBRE := 'JUAN';
  P.DIREC  := DIR;
  P.FECHA_NAC := '10/11/1988' ;
  DBMS_OUTPUT.PUT_LINE('NOMBRE: ' ||P.NOMBRE || ' * CALLE: ' ||
                       P.DIREC.CALLE);
  --
  DIR2 := NEW DIRECCION ('C/Madrid 10', 'Toledo', 45002);
  P2 := NEW PERSONA(2, 'JUAN', DIR2, SYSDATE);
```

```
  DBMS_OUTPUT.PUT_LINE('NOMBRE: ' ||P2.NOMBRE || ' * CALLE: ' ||
                       P2.DIREC.CALLE );
END;
/
```

¡¡INTERESANTE!!

Si utilizamos SQLDEVELOPER, para activar DBMS_OUTPUT seguimos estos pasos: Opción de menú ***Ver->Salida de DBMS***, se abre una nueva ventana. Pulsar el botón + para activar DBMS_OUTPUT y a continuación seleccionar la conexión. Desde la línea de comandos escribimos: SET SERVEROUTPUT ON.

Para borrar un tipo usamos la orden **DROP TYPE** indicando a la derecha el nombre de tipo a borrar: *DROP TYPE nombre_tipo;*

ACTIVIDAD 4.1

Crea un tipo con nombre T_ALUMNO, con 4 atributos, uno de tipo PERSONA y tres que indican las notas de la primera, segunda y tercera evaluación. Después crea un bloque PL/SQL e inicializa un objeto de ese tipo.

4.2.2.1. Métodos

Normalmente cuando creamos un objeto también creamos los métodos que definen el comportamiento del mismo y que permiten actuar sobre él. Los métodos son procedimientos y funciones que se especifican después de los atributos del objeto. Pueden ser de varios tipos:

- **MEMBER**: son los métodos que sirven para actuar con los objetos. Pueden ser procedimientos y funciones.
- **STATIC**: son métodos estáticos independientes de las instancias del objeto. Pueden ser procedimientos y funciones. Estos métodos son operaciones globales que no son de los objetos, sino del tipo.
- **CONSTRUCTOR**: sirve para inicializar el objeto. Se trata de una función cuyos argumentos son los valores de los atributos del objeto y que devuelve el objeto inicializado.

Por cada objeto existe un constructor predefinido por Oracle. Los parámetros del constructor coinciden con los atributos del tipo de objeto, esto es, los parámetros y los atributos se declaran en el mismo orden y tienen el mismo nombre y tipo. No obstante, podemos sobrescribirlo y/o crear otros constructores adicionales; además, creando nuestros propios constructores podemos incluir valores por defecto, restricciones, etc. Los constructores llevarán en la cláusula RETURN la expresión **RETURN SELF AS RESULT**. PL/SQL nunca invoca al constructor implícitamente, por lo que el usuario debe invocarlo explícitamente.

El siguiente ejemplo muestra el tipo DIRECCION con la declaración de un procedimiento que asigna valor al atributo CALLE y una función que devuelve el valor del atributo CALLE (antes de ejecutar el siguiente código hemos de borrar los tipos creados anteriormente con la orden *DROP TYPE nombretipo*):

```
CREATE OR REPLACE TYPE DIRECCION AS OBJECT
(
  CALLE        VARCHAR2(25),
  CIUDAD       VARCHAR2(20),
  CODIGO_POST NUMBER(5),
  MEMBER PROCEDURE SET_CALLE(C VARCHAR2),
```

```
  MEMBER FUNCTION GET_CALLE RETURN VARCHAR2
);
/
```

El siguiente ejemplo define un tipo rectángulo con 3 atributos, un constructor que recibe 2 parámetros, un método **STATIC** y otro **MEMBER**:

```
CREATE OR REPLACE TYPE RECTANGULO AS OBJECT
(
  BASE   NUMBER,
  ALTURA NUMBER,
  AREA   NUMBER,
  STATIC PROCEDURE PROC1 (ANCHO INTEGER, ALTO INTEGER),
  MEMBER PROCEDURE PROC2 (ANCHO INTEGER, ALTO INTEGER),
  CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
                       RETURN SELF AS RESULT
);
/
```

Una vez creado el tipo con la especificación de los atributos y los métodos crearemos el cuerpo del tipo con la implementación de los métodos, usaremos la instrucción **CREATE OR REPLACE TYPE BODY**:

```
CREATE OR REPLACE TYPE BODY nombre_del_tipo AS
<implementación de los métodos>
END;
```

Donde *<implementación de los métodos>* tiene el siguiente formato:

```
[STATIC | MEMBER] PROCEDURE nombreProc [(parametro1, parámetro2, ...)]
IS
  Declaraciones;
BEGIN
  Instrucciones;
END;
[STATIC | MEMBER | CONSTRUCTOR] FUNCTION nombreFunc
[(param1, param2, ... )] RETURN tipo_valor_retorno
IS
  Declaraciones;
BEGIN
  Instrucciones;
END;
```

La implementación de los métodos del objeto DIRECCION es la siguiente:

```
CREATE OR REPLACE TYPE BODY DIRECCION AS
--
  MEMBER PROCEDURE SET_CALLE(C VARCHAR2) IS
  BEGIN
    CALLE := C;
  END;
--
  MEMBER FUNCTION GET_CALLE RETURN VARCHAR2 IS
  BEGIN
```

```
    RETURN CALLE;
  END;
END;
/
```

El siguiente bloque PL/SQL muestra el uso del objeto DIRECCION, visualizará el nombre de la calle, al no definir constructor es necesario invocarlo al definir el objeto (también se puede llamar al constructor con el operador **NEW**):

```
DECLARE
  DIR DIRECCION := DIRECCION(NULL, NULL, NULL);--Llamada al constructor
BEGIN
  DIR.SET_CALLE('La Mina, 3');
  DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
  DIR := NEW DIRECCION ('C/Madrid 10', 'Toledo', 45002);
  DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
END;
/
```

La implementación de los métodos del objeto RECTÁNGULO se muestra a continuación; antes se crea la tabla TABLAREC que usarán los métodos para insertar datos. En el constructor para hacer referencia a los atributos del objeto a partir del cual se invocó el método usamos el cualificador **SELF** delante del atributo, en el método **STATIC** no están permitidas las referencias a los atributos de instancia, en los métodos **MEMBER** sí está permitido:

```
CREATE TABLE TABLAREC (VALOR INTEGER);
/

CREATE OR REPLACE TYPE BODY RECTANGULO AS
--
  CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
                                    RETURN SELF AS RESULT IS
  BEGIN
    SELF.BASE := BASE;
    SELF.ALTURA := ALTURA;
    SELF.AREA := BASE * ALTURA;
    RETURN;
  END;
--
  STATIC PROCEDURE PROC1 (ANCHO INTEGER, ALTO INTEGER) IS
  BEGIN
    INSERT INTO TABLAREC VALUES(ANCHO*ALTO);
    --ALTURA := ALTO; --ERROR NO SE PUEDE ACCEDER A LOS ATRIBUTOS DEL TIPO
    DBMS_OUTPUT.PUT_LINE('FILA INSERTADA');
    COMMIT;
  END;
--
  MEMBER PROCEDURE PROC2 (ANCHO INTEGER, ALTO INTEGER) IS
  BEGIN
    SELF.ALTURA := ALTO; --SE PUEDE ACCEDER A LOS ATRIBUTOS DEL TIPO
    SELF.BASE := ANCHO;
    AREA := ALTURA*BASE;
    INSERT INTO TABLAREC VALUES(AREA);
    DBMS_OUTPUT.PUT_LINE('FILA INSERTADA');
```

```
    COMMIT;
  END;
END;
/
```

El siguiente bloque PL/SQL muestra el uso del objeto RECTANGULO, se puede llamar al constructor usando los 3 atributos; pero es más robusto llamarlo usando 2 atributos, de esta manera nos aseguramos que el atributo AREA tiene el valor inicial correcto. En este caso no es necesario inicializar los objetos R1 y R2, ya que se inicializan en el bloque BEGIN al llamar al constructor con **NEW**:

```
DECLARE
  R1 RECTANGULO;
  R2 RECTANGULO;
  R3 RECTANGULO := RECTANGULO(NULL, NULL, NULL);
BEGIN
  R1 := NEW RECTANGULO(10, 20, 200);
  DBMS_OUTPUT.PUT_LINE('AREA R1: '||R1.AREA);

  R2 := NEW RECTANGULO(10,20);
  DBMS_OUTPUT.PUT_LINE('AREA R2: '||R2.AREA);

  R3.BASE := 5;
  R3.ALTURA := 15;
  R3.AREA := R3.BASE * R3.ALTURA;
  DBMS_OUTPUT.PUT_LINE('AREA R3: '||R3.AREA);

  --USO DE LOS MÉTODOS DEL TIPO  RECTANGULO
  RECTANGULO.PROC1(10, 20);    --LLAMADA AL MÉTODO STATIC
  --RECTANGULO.PROC2(20, 30); --ERROR, LLAMADA AL MÉTODO MEMBER
  --R1.PROC1(5, 6);           --ERROR, LLAMADA AL MÉTODO STATIC
  R1.PROC2(5, 10);            --LLAMADA AL MÉTODO MEMBER
END;
/
```

En cuanto a los métodos, se produce error al llamar al método **STATIC** usando cualquiera de los objetos instanciados. También se produce error si la llamada a un método **MEMBER** se realiza sin haber instanciado un objeto.

Para borrar el cuerpo de un tipo usamos la orden **DROP TYPE BODY** indicando a la derecha el nombre del tipo cuyo cuerpo deseamos borrar: *DROP TYPE BODY nombre_tipo*.

ACTIVIDAD 4.2

Crea un método y el cuerpo del mismo en el tipo T_ALUMNO que devuelva la nota media del alumno.

En muchas ocasiones necesitamos comparar e incluso ordenar datos de tipos definidos como OBJECT. Para ello es necesario crear un método **MAP** u **ORDER**, debiéndose definir al menos uno de ellos por cada objeto que se quiere comparar:

- Los métodos **MAP** consisten en una función que devuelve un valor de tipo escalar (CHAR, VARCHAR2, NUMBER, DATE, etc.) que será el que se utilice en las comparaciones y ordenaciones aplicando los criterios establecidos para este tipo de datos.

- Un método **ORDER** utiliza los atributos del objeto sobre el que se ejecuta para realizar un cálculo y compararlo con otro objeto del mismo tipo que toma como argumento de entrada. Este método devuelve un valor negativo si el parámetro de entrada es mayor que el atributo, un valor positivo si ocurre lo contrario y cero si ambos son iguales. Suelen ser menos funcionales y eficientes, se utilizan cuando el criterio de comparación es muy complejo como para implementarlo con un método **MAP**. No lo trataremos en este Capítulo.

Por ejemplo, la siguiente declaración indica que los objetos de tipo PERSONA se van a comparar por su atributo CODIGO:

```
CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
  CODIGO NUMBER,
  NOMBRE VARCHAR2(35),
  DIREC DIRECCION,
  FECHA_NAC DATE,
  MAP MEMBER FUNCTION POR_CODIGO RETURN NUMBER
);
/
CREATE OR REPLACE TYPE BODY PERSONA AS
  MAP MEMBER FUNCTION POR_CODIGO RETURN NUMBER IS
  BEGIN
    RETURN CODIGO;
  END;
END;
/
```

El siguiente código PL/SQL compara dos objetos de tipo PERSONA, y visualiza '*OBJETOS IGUALES*' ya que el atributo CODIGO tiene el mismo valor para los dos objetos:

```
DECLARE
  P1 PERSONA := PERSONA(NULL, NULL, NULL, NULL);
  P2 PERSONA := PERSONA(NULL, NULL, NULL, NULL);
BEGIN
  P1.CODIGO := 1;
  P1.NOMBRE := 'JUAN';
  P2.CODIGO := 1;
  P2.NOMBRE := 'MANUEL';

  IF P1 = P2 THEN
    DBMS_OUTPUT.PUT_LINE('OBJETOS IGUALES');
  ELSE
    DBMS_OUTPUT.PUT_LINE('OBJETOS DISTINTOS');
  END IF;
END;
/
```

Es necesario un método **MAP** u **ORDER** para comparar objetos en PL/SQL. Un tipo de objeto solo puede tener un método **MAP** o uno **ORDER**.

4.2.3. Tablas de objetos

Una vez definidos los objetos podemos utilizarlos para definir nuevos tipos, para definir columnas de tablas de ese tipo o para definir tablas que almacenan objetos. Una tabla de objetos es una tabla que almacena un objeto en cada fila, se accede a los atributos de esos objetos como si se tratasen de columnas de la tabla. El siguiente ejemplo crea la tabla ALUMNOS de tipo PERSONA con la columna CODIGO como clave primaria y muestra su descripción:

```
CREATE TABLE ALUMNOS OF PERSONA (
  CODIGO PRIMARY KEY
);

DESC ALUMNOS;
Nombre                                    Nulo     Tipo
 ----------------------------------------- -------- ------------------
 CODIGO                                    NOT NULL NUMBER
 NOMBRE                                             VARCHAR2(35)
 DIREC                                              DIRECCION
 FECHA_NAC                                          DATE
```

A continuación se insertan filas en la tabla ALUMNOS. Hemos de poner delante el tipo (DIRECCION) a la hora de dar valores a los atributos que forman la columna de dirección:

```
INSERT INTO ALUMNOS VALUES(
  1, 'Juan Pérez ',
  DIRECCION ('C/Los manantiales 5', 'GUADALAJARA', 19005),
  '18/12/1991'
);

INSERT INTO ALUMNOS (CODIGO, NOMBRE, DIREC, FECHA_NAC) VALUES (
  2, 'Julia Breña',
  DIRECCION ('C/Los espartales 25', 'GUADALAJARA', 19004),
  '18/12/1987'
);
```

El siguiente bloque PL/SQL inserta una fila en la tabla ALUMNOS:

```
DECLARE
  DIR DIRECCION := DIRECCION('C/Sevilla 20', 'GUADALAJARA', 19004);
  PER PERSONA := PERSONA(5, 'MANUEL',DIR, '20/10/1987');
BEGIN
  INSERT INTO ALUMNOS VALUES(PER); --insertar
  COMMIT;
END;
/
```

Veamos algunos ejemplos de consultas sobre la tabla:

- Seleccionar aquellas filas cuya CIUDAD = 'GUADALAJARA':

```
SELECT * FROM ALUMNOS A WHERE A.DIREC.CIUDAD = 'GUADALAJARA';
```

- Para seleccionar columnas individuales, si la columna es un tipo **OBJECT** se necesita definir un alias para la tabla; en una base de datos con tipos y objetos se recomienda usar alias para el nombre de las tablas. A continuación seleccionamos el código y la dirección de los alumnos:

```
SELECT CODIGO, A.DIREC FROM ALUMNOS A;
```

- Para llamar a los métodos hay que utilizar su nombre y paréntesis que encierren los argumentos de entrada (aunque no tenga argumentos los paréntesis deben aparecer). En el siguiente ejemplo obtenemos el nombre y la calle de los alumnos, usamos el método GET_CALLE del tipo DIRECCION:

```
SELECT NOMBRE, A.DIREC.GET_CALLE() FROM ALUMNOS A;
```

- Modificamos aquellas filas cuya ciudad es GUADALAJARA, convertimos la ciudad a minúscula:

```
UPDATE ALUMNOS A
       SET A.DIREC.CIUDAD = LOWER(A.DIREC.CIUDAD)
WHERE A.DIREC.CIUDAD = 'GUADALAJARA';
```

- Eliminamos aquellas filas cuya ciudad sea 'guadalajara':

```
DELETE ALUMNOS A WHERE A.DIREC.CIUDAD = 'guadalajara';
```

- El siguiente bloque PL/SQL muestra el nombre y la calle de los alumnos:

```
DECLARE
  CURSOR C1 IS SELECT * FROM ALUMNOS;
BEGIN
  FOR I IN C1 LOOP
   DBMS_OUTPUT.PUT_LINE(I.NOMBRE || ' - Calle: ' || I.DIREC.CALLE);
  END LOOP;
END;
/
```

- El siguiente bloque PL/SQL modifica la dirección completa de un alumno:

```
DECLARE
  D DIRECCION := DIRECCION ('C/Galiano 5', 'Guadalajara', 19004);
BEGIN
  UPDATE ALUMNOS
         SET DIREC = D WHERE NOMBRE = 'Juan Pérez';
  COMMIT;
END;
/
```

ACTIVIDAD 4.3

Crea la tabla ALUMNOS2 del tipo T_ALUMNO e inserta objetos en ella. Realiza luego una consulta que visualice:

- El nombre del alumno y la nota media.

- Alumnos de GUADALAJARA con nota media mayor de 6.
- Nombre de alumno con más nota media.
- Nombre de alumno con nota más alta (cualquiera de sus notas).

Realiza los ejercicios propuestos 2 y 3.

4.2.4. Tipos colección

Las bases de datos relacionales orientadas a objetos pueden permitir el almacenamiento de colecciones de elementos en una única columna. Tal es el caso de los **VARRAYS** en Oracle que son similares a los arrays de C que permiten almacenar un conjunto de elementos, todos del mismo tipo, y cada elemento tiene un índice asociado; y de las tablas anidadas que permiten almacenar en una columna de una tabla a otra tabla.

4.2.4.1. VARRAYS

Para crear una colección de elementos varrays se usa la orden **CREATE TYPE**. El siguiente ejemplo crea un tipo **VARRAY** de nombre TELEFONO de tres elementos donde cada elemento es del tipo VARCHAR2:

```
CREATE TYPE TELEFONO AS VARRAY(3) OF VARCHAR2(9);
```

Cuando se declara un tipo **VARRAY** no se produce ninguna reserva de espacio. Para obtener información de un **VARRAY** usamos la orden **DESC** (*DESC TELEFONO*). La vista **USER_VARRAYS** obtiene información de las tablas que tienen columnas varrays.

Veamos algunos ejemplos del uso de varrays:

- Creamos una tabla donde una columna es de tipo **VARRAY**:

```
CREATE TABLE AGENDA
(
  NOMBRE VARCHAR2(15),
  TELEF TELEFONO
);
```

- Insertamos varias filas:

```
INSERT INTO AGENDA VALUES
  ('MANUEL', TELEFONO ('656008876', '927986655', '639883300'));
INSERT INTO AGENDA (NOMBRE, TELEF) VALUES
  ('MARTA' , TELEFONO ('649500800'));
```

- En las consultas es imposible poner condiciones sobre los elementos almacenados dentro del **VARRAY**, además, los valores del **VARRAY** solo pueden ser accedidos y recuperados como bloque, no se puede acceder individualmente a los elementos (desde un programa PL/SQL sí se puede). Seleccionamos determinadas columnas:

```
SELECT TELEF FROM AGENDA;
```

- Podemos usar alias para seleccionar las columnas:

```
SELECT A.TELEF FROM AGENDA A;
```

- Modificamos los teléfonos de MARTA:

```
UPDATE AGENDA SET TELEF=TELEFONO('649500800', '659222222')
WHERE NOMBRE = 'MARTA';
```

Desde un programa PL/SQL se puede hacer un bucle para recorrer los elementos del **VARRAY**. El siguiente bloque visualiza los nombres y los teléfonos de la tabla AGENDA, **I.TELEF.COUNT** devuelve el número de elementos del **VARRAY**:

```
DECLARE
  CURSOR C1 IS SELECT * FROM AGENDA;
  CAD VARCHAR2(50);
BEGIN
  FOR I IN C1 LOOP
    DBMS_OUTPUT.PUT_LINE(I.NOMBRE ||
                ', Número de Telefonos: ' ||I.TELEF.COUNT);
    CAD := '*';

    --Recorrer el varray
    FOR J IN 1 .. I.TELEF.COUNT LOOP
      CAD := CAD || I.TELEF(J) || '*';
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(CAD);
  END LOOP;
END;
/
```

Muestra la siguiente salida:

```
MANUEL, Número de Telefonos: 3
*656008876*927986655*639883300*
MARTA, Número de Telefonos:2
*649500800*659222222*
```

El siguiente ejemplo crea un procedimiento almacenado para insertar datos en la tabla AGENDA, a continuación se muestra la llamada al procedimiento:

```
CREATE OR REPLACE PROCEDURE INSERTAR_AGENDA (N VARCHAR2, T TELEFONO) AS
BEGIN
  INSERT INTO AGENDA VALUES (N, T);
END;
/
BEGIN
  INSERTAR_AGENDA('LUIS', TELEFONO('949009977'));
  INSERTAR_AGENDA('MIGUEL', TELEFONO('949004020', '678905400'));
  COMMIT;
END;
/
```

ACTIVIDAD 4.4

Crea una función almacenada que reciba un nombre de la agenda y devuelva el primer teléfono que tenga. Realiza un bloque PL/SQL que haga uso de la función.

La función deberá de controlar si la persona no tiene teléfonos, si no tiene que devuelva un mensaje indicándolo. Controla los posibles errores.

Para obtener información sobre la colección tenemos los siguientes métodos:

Parámetros	Función
COUNT	Devuelve el número de elementos de la colección
EXISTS	Devuelve TRUE si la fila existe
FIRST/LAST	Devuelve el índice del primer y último elemento de la colección.
NEXT/PRIOR	Devuelve el elemento próximo o anterior al actual
LIMIT	Informa del número máximo de elementos que puede contener la colección

Para modificar los elementos de la colección tenemos los siguientes métodos:

Parámetros	Función
DELETE	Elimina todos los elementos de la colección
EXTEND	Añade un elemento nulo a al colección
EXTEND(n)	Añade n elementos nulos
TRIM	Elimina el elemento situado al final de la colección
TRIM(n)	Elimina n elementos del final de la colección

El siguiente ejemplo muestra cómo usar los parámetros:

```
DECLARE
  TEL TELEFONO := TELEFONO(NULL, NULL, NULL);
BEGIN
  SELECT TELEF INTO  TEL FROM AGENDA WHERE NOMBRE = 'MARTA';

  --Visualizar Datos
  DBMS_OUTPUT.PUT_LINE('N° DE TELÉFONOS ACTUALES:    ' || TEL.COUNT);
  DBMS_OUTPUT.PUT_LINE('ÍNDICE DEL PRIMER ELEMENTO: ' || TEL.FIRST);
  DBMS_OUTPUT.PUT_LINE('ÍNDICE DEL ÚLTIMO ELEMENTO: ' || TEL.LAST);
  DBMS_OUTPUT.PUT_LINE('MÁXIMO N° DE TLFS PERMITIDO:' || TEL.LIMIT);

  --Añade un número de teléfono a MARTA
  TEL.EXTEND;
  TEL(TEL.COUNT):= '123000000';
  UPDATE AGENDA A SET A.TELEF = TEL WHERE NOMBRE = 'MARTA';

  --Elimina un teléfono
  SELECT TELEF INTO TEL FROM AGENDA WHERE NOMBRE = 'MANUEL';
  TEL.TRIM;    --Elimina el último elemento del array
  TEL.DELETE; --Elimina todos los elementos
  UPDATE AGENDA A SET A.TELEF = TEL WHERE NOMBRE = 'MANUEL';
END;
/
```

ACTIVIDAD 4.5

Crea un VARRAY de 5 elementos de tipo PERSONA.

Crea después la tabla GRUPOS, con dos columnas: la primera contiene el nombre de grupo de tipo VARCHAR2(15) y la segunda es del tipo definido anteriormente.

Partiendo de las tablas EMPLEADOS y DEPARTAMENTOS llena la tabla GRUPOS. Como nombre de grupo se pondrá el nombre de departamento, como nombre de persona el apellido del empleado, como código la columna EMP_NO y como calle la localidad del departamento. Puedes realizar un procedimiento para ello. Cada fila de la tabla GRUPOS representa un departamento con hasta 5 empleados.

Realiza un bloque PL/SQL que recorra la tabla GRUPOS mostrando por cada departamento el apellido de sus empleados.

Realiza los ejercicios propuestos 4, 5 y 6.

4.2.4.2. Tablas anidadas

Una tabla anidada está formada por un conjunto de elementos, todos del mismo tipo. La tabla anidada está contenida en una columna y el tipo de esta columna debe ser un tipo de objeto existente en la base de datos. Para crear una tabla anidada usamos la orden **CREATE TYPE**. Sintaxis:

```
CREATE TYPE nombre_tipo AS TABLE OF tipo_de_dato;
```

El siguiente ejemplo crea un tipo tabla anidada que almacenará objetos del tipo DIRECCION (creado al principio de la unidad):

```
CREATE TYPE TABLA_ANIDADA AS TABLE OF DIRECCION;
```

No es necesario especificar el tamaño máximo de una tabla anidada. Veamos cómo se define una columna de una tabla con el tipo tabla anidada creada anteriormente

```
CREATE TABLE EJEMPLO_TABLA_ANIDADA
(
  ID NUMBER(2),
  APELLIDOS VARCHAR2(35),
  DIREC  TABLA_ANIDADA
)
NESTED TABLE DIREC STORE AS DIREC_ANIDADA;
```

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 (DIREC_ANIDADA) en la que se van a almacenar las direcciones que se representan en el atributo DIREC de cualquier objeto de la tabla EJEMPLO_TABLA_ANIDADA. La descripción del tipo TABLA_ANIDADA y de la tabla EJEMPLO_TABLA_ANIDADA es la siguiente:

```
SQL> DESC TABLA_ANIDADA;
TABLA_ANIDADA TABLE OF DIRECCION
Nombre                                    Nulo    Tipo
 ----------------------------------------- ----- --------------------
 CALLE                                             VARCHAR2(25)
 CIUDAD                                            VARCHAR2(20)
```

```
 CODIGO_POST                                          NUMBER(5)

METHOD
------
MEMBER PROCEDURE SET_CALLE
Nombre de Argumento          Tipo                     E/S     Por Defecto
 ------------------------------ ---------------------- ------ ----------
C                               VARCHAR2                 IN
METHOD
MEMBER FUNCTION GET_CALLE RETURNS VARCHAR2

SQL> DESC EJEMPLO_TABLA_ANIDADA;
Nombre                                        Nulo    Tipo
 ----------------------------------------- ------ --------------------
 ID                                                   NUMBER(2)
 APELLIDOS                                            VARCHAR2(35)
 DIREC                                                TABLA_ANIDADA
```

Veamos algunos ejemplos con la tabla.

- Insertamos varias filas con varias direcciones en la tabla EJEMPLO_TABLA_ANIDADA:

```
INSERT INTO EJEMPLO_TABLA_ANIDADA VALUES (1, 'RAMOS',
  TABLA_ANIDADA (
    DIRECCION ('C/Los manantiales 5', 'GUADALAJARA', 19004),
    DIRECCION ('C/Los manantiales 10', 'GUADALAJARA', 19004),
    DIRECCION ('C/Av de Paris 25', 'CÁCERES ', 10005),
    DIRECCION ('C/Segovia 23-3A', 'TOLEDO', 45005)
  )
);

INSERT INTO EJEMPLO_TABLA_ANIDADA VALUES (2, 'MARTÍN',
  TABLA_ANIDADA (
    DIRECCION ('C/Huesca 5', 'ALCALÁ DE H', 28804),
    DIRECCION ('C/Madrid 20', 'ALCORCÓN', 28921)
  )
);
```

- Se inserta el código, el nombre y la tabla anidada vacía:

```
INSERT INTO EJEMPLO_TABLA_ANIDADA
            VALUES (5, 'PEREZ', TABLA_ANIDADA());
```

- Seleccionamos todas las filas de la tabla:

```
SELECT * FROM EJEMPLO_TABLA_ANIDADA;
```

```
    ID APELLIDOS
---------- ----------
1 RAMOS  EJEMPLO.TABLA_ANIDADA([EJEMPLO.DIRECCION], [EJEMPLO.DIRECCION], [EJEMPLO.DIRECCION], [EJEMPLO.DI
2 MARTÍN EJEMPLO.TABLA_ANIDADA([EJEMPLO.DIRECCION], [EJEMPLO.DIRECCION])
5 PEREZ  EJEMPLO.TABLA_ANIDADA()
```

- El siguiente ejemplo obtiene el identificador, el apellido y la dirección completa de todas las filas de la tabla. Se obtienen tantas filas como calles tiene cada identificador. El operador **TABLE** con la columna que es tabla anidada entre paréntesis y colocado

a la derecha de FROM se utiliza para acceder a todas las filas de la tabla anidada, es necesario indicar el alias (en este caso se llama DIRECCION); con DIRECCION.* se obtienen todos los campos de la dirección:

```
SELECT  ID, APELLIDOS, DIRECCION.*
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) DIRECCION;
```

	ID	APELLIDOS	CALLE	CIUDAD	CODIGO_POST
1	1	RAMOS	C/Los manantiales 5	GUADALAJARA	19004
2	1	RAMOS	C/Los manantiales 10	GUADALAJARA	19004
3	1	RAMOS	C/Av de Paris 25	CÁCERES	10005
4	1	RAMOS	C/Segovia 23-3A	TOLEDO	45005
5	2	MARTÍN	C/Huesca 5	ALCALÁ DE H	28804
6	2	MARTÍN	C/Madrid 20	ALCORCÓN	28921

En la consulta anterior la columna que es tabla anidada se utiliza como si fuese una tabla normal, incluyéndola en la cláusula FROM.

- El siguiente ejemplo obtiene las direcciones completas del identificador 1:

```
SELECT  ID, DIRECCION.*
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) DIRECCION WHERE ID=1;
```

	ID	CALLE	CIUDAD	CODIGO_POST
1	1	C/Los manantiales 5	GUADALAJARA	19004
2	1	C/Los manantiales 10	GUADALAJARA	19004
3	1	C/Av de Paris 25	CÁCERES	10005
4	1	C/Segovia 23-3A	TOLEDO	45005

Se pueden usar cursores dentro de una SELECT para acceder o poner condiciones a las filas de una tabla anidada. La sintaxis es la siguiente:

```
CURSOR (SELECT columnas FROM TABLE (columna_tabla_anidada))
```

Donde ***columnas*** son las columnas del tipo de dato de la tabla anidada.

- A continuación obtenemos el identificador, los apellidos y las calles y ciudad de cada fila de la tabla, se obtienen tantas filas como filas hay en la tabla, con el operador **TABLE** se hace referencia a la tabla anidada:

```
SELECT ID, APELLIDOS, CURSOR (SELECT CALLE, CIUDAD FROM TABLE(DIREC))
FROM EJEMPLO_TABLA_ANIDADA ;
```

	ID	APELLIDOS	CURSOR(SELECTCALLE,CIUDADFROMTABLE(DIREC))
1	1	RAMOS	{<CALLE=C/Los manantiales 5,CIUDAD=GUADALAJARA>,<CALLE=C/Los manantiales 10,CIUDAD=
2	2	MARTÍN	{<CALLE=C/Huesca 5,CIUDAD=ALCALÁ DE H>,<CALLE=C/Madrid 20,CIUDAD=ALCORCÓN>,}
3	5	PEREZ	{}

- Es habitual el uso de alias en las tablas anidadas:

```
SELECT ID, APELLIDOS,
       CURSOR (SELECT T.CALLE, T.CIUDAD FROM TABLE(DIREC) T)
FROM EJEMPLO_TABLA_ANIDADA ;
```

- Las siguientes consultas muestran el número de direcciones de cada identificador, la primera usa la tabla anidada dentro de un **CURSOR**, la segunda como si fuese una

tabla normal utilizando el operador **TABLE**, en este caso es necesario usar GROUP BY, ya que cada identificador puede tener varias direcciones:

```
SELECT ID, APELLIDOS, CURSOR(SELECT count(*) FROM TABLE(DIREC) )
FROM EJEMPLO_TABLA_ANIDADA;
```

	ID	APELLIDOS	CURSOR(SELECTCOUNT(*)FROMTABLE(DIREC))
1	1	RAMOS	{<COUNT(*)=4>,}
2	2	MARTÍN	{<COUNT(*)=2>,}
3	5	PEREZ	{<COUNT(*)=0>,}

```
SELECT ID, APELLIDOS, count(*)
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC)
GROUP BY ID, APELLIDOS;
```

	ID	APELLIDOS	COUNT(*)
1	1	RAMOS	4
2	2	MARTÍN	2

- Las siguientes consultas muestran aquellas filas que tienen 2 direcciones en la CIUDAD de GUADALAJARA, la primera usa CURSOR para acceder a la tabla anidada y la segunda usa la tabla anidada como si fuese una tabla normal, que se combina con otra tabla:

```
SELECT ID, APELLIDOS, CURSOR (SELECT COUNT(*) FROM  TABLE(DIREC)
                              WHERE CIUDAD = 'GUADALAJARA')
FROM EJEMPLO_TABLA_ANIDADA
WHERE
  (SELECT COUNT(*)FROM TABLE(DIREC) WHERE CIUDAD = 'GUADALAJARA') = 2;
```

	ID	APELLIDOS	CURSOR(SELECTCOUNT(*)FROMTABLE(DIREC)WHERECIUDAD='GUADALAJARA')
1	1	RAMOS	{<COUNT(*)=2>,}

```
SELECT ID, APELLIDOS, COUNT(*)
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC)
WHERE CIUDAD = 'GUADALAJARA'
GROUP BY ID, APELLIDOS HAVING COUNT(*) = 2;
```

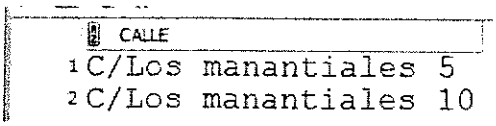
	ID	APELLIDOS	COUNT(*)
1	1	RAMOS	2

Para seleccionar filas de una tabla anidada se puede utilizar la cláusula **THE** con SELECT. La sintaxis es:

```
SELECT ... FROM THE (subconsulta sobre tabla anidada) WHERE ...
```

- El siguiente ejemplo obtiene las calles de la fila con ID = 1 cuya ciudad sea GUADALAJARA, se obtienen tantas filas como calles hay en la ciudad de GUADALAJARA:

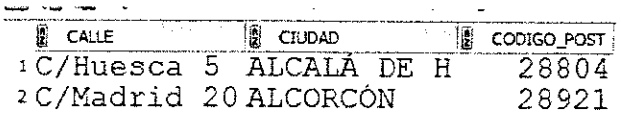
```
SELECT CALLE FROM THE
            (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1)
WHERE CIUDAD = 'GUADALAJARA';
```



	CALLE
1	C/Los manantiales 5
2	C/Los manantiales 10

- La siguiente consulta obtiene todos los datos de las direcciones del identificador 2:

```
SELECT * FROM THE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 2);
```



	CALLE	CIUDAD	CODIGO_POST
1	C/Huesca 5	ALCALÁ DE H	28804
2	C/Madrid 20	ALCORCÓN	28921

- La siguiente consulta usa la tabla anidada en FROM y obtiene el mismo resultado que la anterior:

```
SELECT TT.* FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) TT WHERE ID = 2;
```

ACTIVIDAD 4.6

Obtén el número de direcciones que tiene en cada ciudad el identificador 1.

Obtén la ciudad con más direcciones que tiene el identificador 1.

Realiza un bloque PL/SQL que muestre el nombre de las calles de cada apellido.

- Insertamos una dirección al final de la tabla anidada para el identificador 1 (ahora el identificador 1 tendrá cinco direcciones):

```
INSERT INTO TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1)
VALUES (DIRECCION ('C/Los manantiales 15', 'GUADALAJARA', 19004));
```

La cláusula **TABLE** a la derecha de INTO se utiliza para acceder a la fila que nos interesa, en este caso la que tiene ID = 1.

- En el siguiente ejemplo se modifica la primera dirección del identificador 1, se le asigna el valor *'C/Pilón 11', 'TOLEDO', 45589*:

```
UPDATE TABLE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET VALUE(PRIMERA) = DIRECCION ('C/Pilón 11', 'TOLEDO', 45589)
WHERE
VALUE(PRIMERA)= DIRECCION('C/Los manantiales 5', 'GUADALAJARA', 19004);
```

El alias **PRIMERA** recoge los datos devueltos por la SELECT (que debe devolver una fila). Con **SET VALUE (PRIMERA)** se asigna el valor *'C/Pilón 11', 'TOLEDO', 45589* al objeto DIRECCIÓN cuyo valor coincida con *'C/Los manantiales 5', 'GUADALAJARA', 19004*; esto se indica en la cláusula WHERE con la función **VALUE(PRIMERA)**.

- En el siguiente ejemplo se modifican (para el identificador 1) todas las direcciones que tengan la ciudad de GUADALAJARA, se le asigna el valor MADRID. En este caso no se necesita la función **VALUE**, ya que se modifica la columna CIUDAD y no un objeto:

```
UPDATE TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET    PRIMERA.CIUDAD = 'MADRID'
WHERE PRIMERA.CIUDAD = 'GUADALAJARA';
```

En el siguiente ejemplo se elimina la segunda dirección del identificador 1, aquella cuyo valor es *'C/Los manantiales 10', 'GUADALAJARA', 19004*:

```
DELETE FROM TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los manantiales 10', 'GUADALAJARA', 19004);
```

En el siguiente ejemplo se eliminan todas las direcciones del identificador 1 con ciudad igual a 'GUADALAJARA':

```
DELETE FROM TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE PRIMERA.CIUDAD = 'GUADALAJARA';
```

- El siguiente bloque PL/SQL crea un procedimiento que recibe un identificador y visualiza las calles que tiene, debajo se muestra el bloque PL/SQL que prueba el procedimiento:

```
CREATE OR REPLACE PROCEDURE VER_DIREC(IDENT NUMBER) AS
  CURSOR C1 IS
       SELECT CALLE FROM THE
      (SELECT T.DIREC FROM EJEMPLO_TABLA_ANIDADA T WHERE ID = IDENT);
BEGIN
  FOR I IN C1 LOOP
    DBMS_OUTPUT.PUT_LINE(I.CALLE);
  END LOOP;
END VER_DIREC;
/
--Probando el procedimiento
BEGIN
  VER_DIREC(1);
END;
/
```

La vista **USER_NESTED_TABLES** obtiene información de las tablas anidadas.

- El siguiente ejemplo crea una función almacenada que comprueba si existe una dirección en un identificador concreto. La función recibe el identificador y un tipo DIRECCION, devuelve un mensaje indicando si existe o no la dirección. Primero se comprobará si existe el identificador, si no existe o si existen varias filas con el mismo identificador se devuelve un mensaje indicándolo.

```
CREATE OR REPLACE FUNCTION EXISTE_DIREC
                          (IDEN NUMBER, DIR DIRECCION)
         RETURN VARCHAR2 AS
  IDT NUMBER;
  CUENTA NUMBER;
BEGIN
```

```
--COMPROBAR SI EXISTE ID:
  SELECT COUNT(ID) INTO CUENTA
  FROM EJEMPLO_TABLA_ANIDADA WHERE ID = IDEN;

  IF CUENTA = 0 THEN
    RETURN 'NO EXISTE EL ID: '|| IDEN || ', EN LA TABLA';
  END IF;

  IF CUENTA > 1 THEN
    RETURN 'EXISTEN VARIOS REGISTROS CON EL MISMO  ID: ' ||IDEN;
  END IF;

--EL ID EXISTE, COMPROBAR SI LA CALLE EXISTE:
  SELECT ID INTO IDT
         FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC)
         WHERE ID = IDEN
         AND UPPER(CALLE) = UPPER(DIR.CALLE)
         AND UPPER(CIUDAD) = UPPER(DIR.CIUDAD)
         AND CODIGO_POST = DIR.CODIGO_POST;

  RETURN ('LA DIRECCIÓN : '||DIR.CALLE || '*' ||DIR.CIUDAD
                           || '*' || DIR.CODIGO_POST
                           || ' YA EXISTE PARA ESE ID: '||IDEN);

EXCEPTION
  WHEN NO_DATA_FOUND THEN
  RETURN 'NO EXISTE LA DIRECCIÓN : '||DIR.CALLE || '*'  ||DIR.CIUDAD
         || '*' || DIR.CODIGO_POST   || ' PARA EL ID: ' ||IDEN;
END EXISTE_DIREC;
/

--Probando la función
BEGIN
  DBMS_OUTPUT.PUT_LINE
    (EXISTE_DIREC(1, DIRECCION('C/Huesca 5', 'ALCALÁ DE H', 28804)));
  DBMS_OUTPUT.PUT_LINE
    (EXISTE_DIREC(2, DIRECCION('C/Huesca 5', 'ALCALÁ DE H', 28804)));
END;
/
```

ACTIVIDAD 4.7

Realiza un procedimiento almacenado para insertar direcciones en la tabla EJEMPLO_TABLA_ANIDADA.

El procedimiento recibe como parámetros un identificador y un objeto DIRECCION. Debe visualizar un mensaje indicando si se ha insertado o no la dirección.

Se deben hacer las siguientes comprobaciones y visualizar los mensajes correspondientes:

- Comprobar si el identificador existe, si no existe es un caso de error, visualizar mensaje.
- Que la tabla anidada no sea null, si es null hay que hacer un UPDATE no un INSERT.
- Que la dirección no exista ya en la tabla, si ya existe visualiza que no se puede insertar.

4.2.5. Referencias

Mediante el operador **REF** asociado a un atributo se pueden definir referencias a otros objetos. Un atributo de este tipo almacena una referencia al objeto del tipo definido e implementa una relación de asociación entre los dos tipos de objetos. Una columna de tipo **REF** guarda un puntero a una fila de la otra tabla, contiene el OID (identificador del objeto fila) de dicha fila. Ejemplos:

- El siguiente ejemplo crea un tipo EMPLEADO_T donde uno de los atributos es una referencia a un objeto EMPLEADO_T, después se crea una tabla de objetos EMPLEADO_T:

```
CREATE TYPE EMPLEADO_T AS OBJECT (
  NOMBRE      VARCHAR2(30),
  JEFE        REF EMPLEADO_T
);
/
CREATE TABLE EMPLEADO OF EMPLEADO_T;
```

- Insertamos filas en la tabla, el segundo INSERT asigna al atributo JEFE la referencia al objeto con apellido GIL:

```
INSERT INTO EMPLEADO VALUES (EMPLEADO_T ('GIL', NULL));

INSERT INTO EMPLEADO SELECT EMPLEADO_T ('ARROYO', REF(E))
FROM EMPLEADO E WHERE E.NOMBRE = 'GIL';

INSERT INTO EMPLEADO SELECT EMPLEADO_T ('RAMOS', REF(E))
FROM EMPLEADO E WHERE E.NOMBRE = 'GIL';
```

Para acceder al objeto referido por un **REF** se utiliza el operador **DEREF**, en el ejemplo se visualiza el nombre del empleado y los datos del jefe de cada empleado:

```
SQL> SELECT NOMBRE, DEREF(P.JEFE) FROM EMPLEADO P;
NOMBRE          DEREF(P.JEFE)(NOMBRE, JEFE)
--------------- ------------------------------------------------------
GIL
ARROYO          EMPLEADO_T('GIL', NULL)
RAMOS           EMPLEADO_T('GIL', NULL)
```

- La siguiente consulta obtiene el identificador del objeto cuyo nombre es GIL:

```
SELECT REF(P) FROM EMPLEADO P WHERE NOMBRE = 'GIL';
```

- La siguiente consulta obtiene nombre del empleado y el nombre de su jefe:

```
SQL> SELECT NOMBRE, DEREF(P.JEFE).NOMBRE FROM EMPLEADO P;
NOMBRE                         DEREF(P.JEFE).NOMBRE
------------------------------ ------------------------------
GIL
ARROYO                         GIL
RAMOS                          GIL
```

- El siguiente ejemplo actualiza el jefe del nombre RAMOS, se le asigna ARROYO:

```
UPDATE EMPLEADO
  SET JEFE = (SELECT REF(E) FROM EMPLEADO E WHERE NOMBRE = 'ARROYO')
WHERE NOMBRE = 'RAMOS '.
```

ACTIVIDAD 4.8

Crea un TIPO_DEP con las siguientes columnas: DEPT_NO NUMBER(2), DNOMBRE VARCHAR2(15), LOC VARCHAR2(15). Crea una tabla del tipo definido anteriormente. Llena la tabla a partir de los datos de la tabla DEPARTAMENTOS.

Crea una tabla con las siguientes columnas, una de ellas es una referencia a un TIPO_DEP: EMP_NO NUMBER(4), APELLIDO VARCHAR2(15), SALARIO NUMBER(6,2) y DEPT REF TIPO_DEP. Llena esta tabla a partir de la tabla EMPLEADOS.

Haz un bloque PL/SQL que recorra esta última tabla y muestre el apellido, salario, número de departamento, nombre y localidad.

4.2.6. Herencia de tipos

La herencia facilita la creación de objetos a partir de otros ya existentes e implica que un subtipo obtenga todo el comportamiento (métodos) y eventualmente los atributos de su supertipo. Los subtipos definen sus propios atributos y métodos y puede redefinir los métodos que heredan, esto se conoce como polimorfismo. El siguiente ejemplo define un tipo persona y a continuación el subtipo tipo alumno:

```
--Se define el tipo persona
--
CREATE OR REPLACE TYPE TIPO_PERSONA AS OBJECT(
  DNI VARCHAR2(10),
  NOMBRE VARCHAR2(25),
  FEC_NAC DATE,
  MEMBER FUNCTION EDAD RETURN NUMBER,
  FINAL MEMBER FUNCTION GET_DNI
                        RETURN VARCHAR2, -- No se puede redefinir
  MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2,
  MEMBER PROCEDURE VER_DATOS
) NOT FINAL;  -- Se pueden derivar subtipos
/
--Cuerpo del tipo persona
--
CREATE OR REPLACE TYPE BODY TIPO_PERSONA AS
  MEMBER FUNCTION EDAD RETURN NUMBER IS
    ED NUMBER;
  BEGIN
    ED := TO_CHAR(SYSDATE, 'YYYY') - TO_CHAR(FEC_NAC, 'YYYY');
    RETURN ED;
  END;
  --
  FINAL MEMBER FUNCTION GET_DNI RETURN VARCHAR2 IS
  BEGIN
    RETURN DNI;
  END;
  --
  MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2 IS
```

```
  BEGIN
    RETURN NOMBRE;
  END;
  --
  MEMBER PROCEDURE VER_DATOS IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(DNI|| '*' || NOMBRE ||'*'||EDAD());
  END;
END;
/
--Se define el tipo alumno
--
CREATE OR REPLACE TYPE TIPO_ALUMNO UNDER TIPO_PERSONA(
                                          --se define un subtipo
  CURSO VARCHAR2(10),
  NOTA_FINAL NUMBER,
  MEMBER FUNCTION NOTA RETURN NUMBER,
  OVERRIDING MEMBER PROCEDURE VER_DATOS  --se redefine ese método
);
/
--Cuerpo del tipo alumno
--
CREATE OR REPLACE TYPE BODY  TIPO_ALUMNO AS
  MEMBER FUNCTION NOTA RETURN NUMBER IS
  BEGIN
    RETURN NOTA_FINAL;
  END;
  --
  OVERRIDING MEMBER PROCEDURE VER_DATOS IS  --se redefine ese método
  BEGIN
    DBMS_OUTPUT.PUT_LINE(CURSO|| '*' ||NOTA_FINAL);
  END;
END;
/
```

Mediante la cláusula **NOT FINAL** (incluida al final de la definición del tipo) se indica que se pueden derivar subtipos, si no se incluye esta cláusula se considera que es **FINAL** (no puede tener subtipos). Igualmente si un método es **FINAL** los subtipos no pueden redefinirlo. La cláusula **OVERRIDING** se utiliza para redefinir el método. El siguiente bloque PL/SQL muestra un ejemplo de uso de los tipos definidos, al definir el objeto se inicializan todos los atributos, ya que no se ha definido constructor para inicializar el objeto:

```
DECLARE
  --Al asignar datos al alumno escribimos
  --         DNI, NOMBRE, FECHA_NAC, CURSO, NOTA

  A1 TIPO_ALUMNO := TIPO_ALUMNO(NULL, NULL, NULL, NULL, NULL);
  A2 TIPO_ALUMNO := TIPO_ALUMNO('871234533A', 'PEDRO',
                                '12/12/1996', 'SEGUNDO', 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 := '20/10/1997';
   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 realiza alguna consulta (al insertar se escriben las columnas del supertipo - dni, nombre, fec_nac - y luego las del subtipo - curso, nota_final -):

```
CREATE TABLE TALUMNOS OF TIPO_ALUMNO (DNI PRIMARY KEY);

INSERT INTO TALUMNOS VALUES
      ('871234533A', 'PEDRO', '12/12/1996', 'SEGUNDO', 7);
INSERT INTO TALUMNOS VALUES
      ('809004534B', 'MANUEL', '12/12/1997', '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;
```

4.2.7. Ejemplo de modelo relacional y objeto relacional

A continuación vamos a ver una solución con el modelo relacional para gestión de ventas y otra usando el enfoque objeto-relacional. En la Figura 4.1 se muestra el modelo de datos para las tablas CLIENTES, PRODUCTOS, VENTAS y LINEASVENTAS. Las órdenes de creación de las tablas son:

```
CREATE TABLE CLIENTES (
   IDCLIENTE NUMBER PRIMARY KEY,
   NOMBRE      VARCHAR2(50),
   DIRECCION   VARCHAR2(50),
   POBLACION   VARCHAR2(50),
   CODPOSTAL   NUMBER(5),
   PROVINCIA   VARCHAR2(40),
   NIF         VARCHAR2(9) UNIQUE,
   TELEFONO1   VARCHAR2(15),
   TELEFONO2   VARCHAR2(15),
   TELEFONO3   VARCHAR2(15)
);

CREATE TABLE PRODUCTOS (
   IDPRODUCTO  NUMBER PRIMARY KEY,
   DESCRIPCION varchar2(80),
   PVP   NUMBER,
   STOCKACTUAL NUMBER
);
```

```
CREATE TABLE VENTAS(
   IDVENTA     NUMBER PRIMARY KEY,
   IDCLIENTE   NUMBER NOT NULL
               REFERENCES CLIENTES,
   FECHAVENTA  DATE
);

CREATE TABLE LINEASVENTAS (
   IDVENTA      NUMBER,
   NUMEROLINEA  NUMBER,
   IDPRODUCTO   NUMBER,
   CANTIDAD     NUMBER,
   FOREIGN KEY (IDVENTA)
      REFERENCES VENTAS (IDVENTA),
   FOREIGN KEY (IDPRODUCTO)
      REFERENCES PRODUCTOS (IDPRODUCTO),
   PRIMARY KEY (IDVENTA,NUMEROLINEA)
);
```

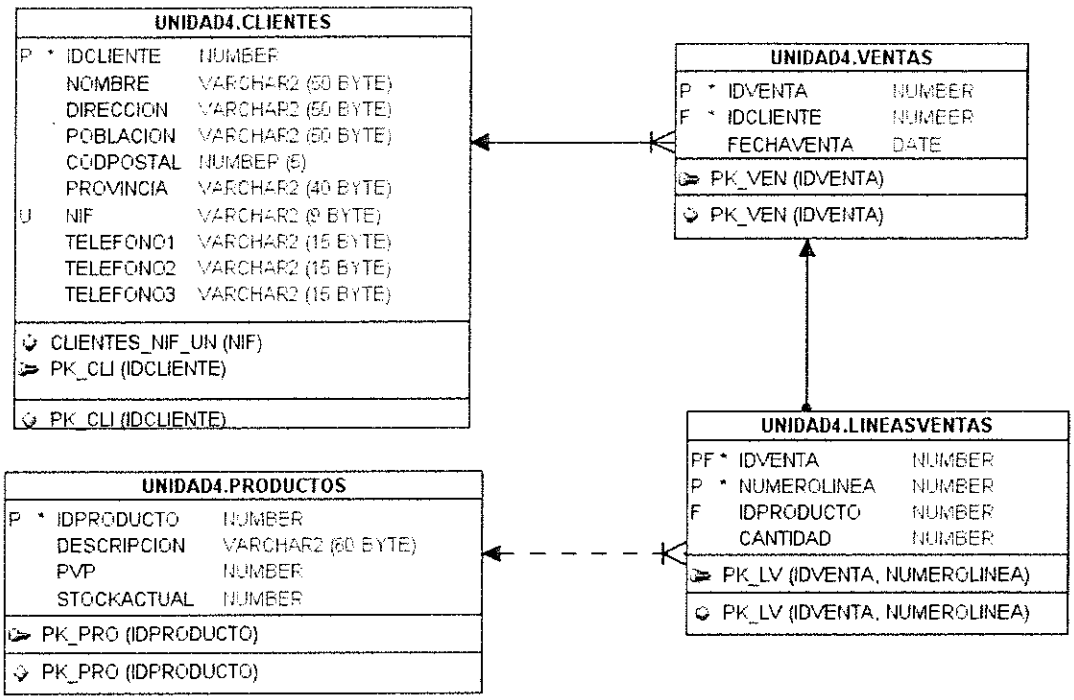


Figura 4.1. Modelo de datos.

Definimos los siguientes tipos:

- Definimos un tipo VARRAY de 3 elementos para contener los teléfonos:

```
CREATE TYPE TIP_TELEFONOS AS VARRAY(3) OF VARCHAR2(15);
/
```

- A continuación se crean los tipos dirección, cliente, producto y línea de venta:

```
CREATE TYPE TIP_DIRECCION AS OBJECT(
  CALLE       VARCHAR2(50),
  POBLACION   VARCHAR2(50),
  CODPOSTAL   NUMBER(5),
  PROVINCIA   VARCHAR2(40)
);
/
CREATE TYPE TIP_CLIENTE AS OBJECT(
  IDCLIENTE   NUMBER,
  NOMBRE      VARCHAR2(50),
  DIREC       TIP_DIRECCION,
  NIF         VARCHAR2(9),
  TELEF       TIP_TELEFONOS
);
/
CREATE TYPE TIP_PRODUCTO AS OBJECT (
  IDPRODUCTO    NUMBER,
  DESCRIPCION   VARCHAR2(80),
  PVP           NUMBER,
  STOCKACTUAL   NUMBER
);
/
```

```
CREATE TYPE  TIP_LINEAVENTA AS OBJECT (
  NUMEROLINEA  NUMBER,
  IDPRODUCTO   REF TIP_PRODUCTO,
  CANTIDAD     NUMBER
);
/
```

- Creamos un tipo tabla anidada para contener las líneas de una venta:

```
CREATE TYPE TIP_LINEAS_VENTA AS TABLE OF TIP_LINEAVENTA;
/
```

- Creamos un tipo venta para los datos de las ventas, cada venta tendrá un atributo LINEAS del tipo tabla anidada definida anteriormente:

```
CREATE TYPE  TIP_VENTA AS OBJECT (
  IDVENTA      NUMBER,
  IDCLIENTE    REF TIP_CLIENTE,
  FECHAVENTA   DATE,
  LINEAS       TIP_LINEAS_VENTA,
  MEMBER FUNCTION TOTAL_VENTA RETURN NUMBER
);
/
```

En el tipo TIP_VENTA se ha definido la función miembro TOTAL_VENTA que calcula el total de la venta de las líneas de venta que forman parte de una venta. COUNT cuenta el número de elementos de una tabla o de un array, **LINEAS.COUNT** devuelve el número de líneas que tiene la venta.

```
CREATE OR REPLACE TYPE BODY TIP_VENTA AS
  MEMBER FUNCTION TOTAL_VENTA RETURN NUMBER IS
    TOTAL NUMBER := 0;
    LINEA   TIP_LINEAVENTA;
    PRODUCT TIP_PRODUCTO;
  BEGIN
    FOR I IN 1..LINEAS.COUNT LOOP
      LINEA := LINEAS(I);
      SELECT DEREF(LINEA.IDPRODUCTO) INTO PRODUCT FROM DUAL;
      TOTAL := TOTAL + LINEA.CANTIDAD * PRODUCT.PVP;
    END LOOP;
    RETURN TOTAL;
  END;
END;
/
```

Creamos las tablas donde almacenar los objetos de la aplicación, la tabla para los clientes, los productos y las ventas, también se definen las claves primarias de dichas tablas:

```
CREATE TABLE TABLA_CLIENTES OF TIP_CLIENTE (
  IDCLIENTE PRIMARY KEY,
  NIF UNIQUE
);
/
CREATE TABLE TABLA_PRODUCTOS OF TIP_PRODUCTO (
```

```
  IDPRODUCTO PRIMARY KEY
);
/
CREATE TABLE TABLA_VENTAS OF TIP_VENTA (
  IDVENTA PRIMARY KEY
) NESTED TABLE LINEAS STORE AS TABLA_LINEAS;
/
```

En la tabla TABLA_VENTAS se define una tabla anidada para el atributo LINEAS del tipo TIP_VENTA, contendrá las líneas de venta.

Insertamos 2 clientes y 5 productos:

```
INSERT INTO TABLA_CLIENTES VALUES
(1, 'Luis Gracia', TIP_DIRECCION('C/Las Flores 23', 'Guadalajara',
            '19003', 'Guadalajara'),
            '34343434L', TIP_TELEFONOS('949876655', '949876655')
);

INSERT INTO TABLA_CLIENTES VALUES
(2, 'Ana Serrano', TIP_DIRECCION ('C/Galiana 6', 'Guadalajara',
            '19004', 'Guadalajara'),
            '76767667F', TIP_TELEFONOS('94980009')
);

INSERT INTO TABLA_PRODUCTOS VALUES
       (1, 'CAJA DE CRISTAL DE MURANO', 100, 5);
INSERT INTO TABLA_PRODUCTOS VALUES (2, 'BICICLETA CITY', 120, 15);
INSERT INTO TABLA_PRODUCTOS VALUES (3, '100 LÁPICES DE COLORES', 20,5);
INSERT INTO TABLA_PRODUCTOS VALUES (4, 'OPERACIONES CON BD', 25, 5);
INSERT INTO TABLA_PRODUCTOS VALUES (5, 'APLICACIONES WEB', 25.50, 10);
```

Insertamos en TABLA_VENTAS la venta con IDVENTA 1 para el IDCLIENTE 1:

```
INSERT INTO TABLA_VENTAS
       SELECT 1, REF(C), SYSDATE, TIP_LINEAS_VENTA()
FROM TABLA_CLIENTES C WHERE C.IDCLIENTE = 1;
```

Insertamos en TABLA_VENTAS dos líneas de venta para el IDVENTA 1 para los productos 1 (la CANTIDAD es 1) y 2 (la CANTIDAD es 2):

```
INSERT INTO TABLE
    (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA = 1)
(SELECT 1, REF(P), 1 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO = 1);

INSERT INTO TABLE
    (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA = 1)
(SELECT 2, REF(P), 2 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO = 2);
```

Insertamos en TABLA_VENTAS la venta con IDVENTA 2 para el IDCLIENTE 1:

```
INSERT INTO TABLA_VENTAS
    SELECT 2, REF(C), SYSDATE, TIP_LINEAS_VENTA()
FROM TABLA_CLIENTES C WHERE C.IDCLIENTE = 1;
```

Insertamos en TABLA_VENTAS tres líneas de venta para el IDVENTA 2 para los productos 1 (la CANTIDAD es 2), 4 (la CANTIDAD es 1) y 5 (la CANTIDAD es 4):

```
INSERT INTO TABLE
    (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA = 2)
(SELECT 1, REF(P), 2 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO = 1);

INSERT INTO TABLE
    (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA = 2)
(SELECT 2, REF(P), 1 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO = 4);

INSERT INTO TABLE
    (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA = 2)
(SELECT 3, REF(P), 4 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO = 5);
```

La siguiente consulta muestra el total de ventas en cada venta:

```
SELECT  IDVENTA, DEREF(IDCLIENTE).NOMBRE NOMBRE,
DEREF(IDCLIENTE).IDCLIENTE IDCLIENTE, T.TOTAL_VENTA() TOTAL
FROM TABLA_VENTAS T;
```

```
IDVENTA NOMBRE                                IDCLIENTE      TOTAL
---------- ------------------------------ ---------- ----------
         1 Luis Gracia                             1        340
         2 Luis Gracia                             1        327
```

La siguiente consulta muestra el detalle de los productos junto con la venta y el cliente; se puede utilizar la tabla anidada como tabla en la consulta poniendo la cláusula **TABLE**:

```
SELECT P.IDVENTA IDV, DEREF(P.IDCLIENTE).NOMBRE NOMBRE,
       DETALLE.NUMEROLINEA LINEA,
       DEREF(DETALLE.IDPRODUCTO).DESCRIPCION PRODUCTO,
       DETALLE.CANTIDAD,
       DETALLE.CANTIDAD * DEREF(DETALLE.IDPRODUCTO).PVP IMPORTE,
       DEREF(DETALLE.IDPRODUCTO).PVP PVP,
       DEREF(DETALLE.IDPRODUCTO).STOCKACTUAL STOCK
FROM TABLA_VENTAS P, TABLE(P.LINEAS) DETALLE;
```

IDV	NOMBRE	LINEA	PRODUCTO	CANTIDAD	IMPORTE	PVP	STOCK
1	Luis Gracia	1	CAJA DE CRISTAL DE MURANO	1	100	100	5
1	Luis Gracia	2	BICICLETA CITY	2	240	120	15
2	Luis Gracia	1	CAJA DE CRISTAL DE MURANO	2	200	100	5
2	Luis Gracia	2	OPERACIONES CON BD	1	25	25	5
2	Luis Gracia	3	APLICACIONES WEB	4	102	25,5	10

El siguiente procedimiento almacenado visualiza los datos de la venta cuyo identificador recibe:

```
CREATE OR REPLACE PROCEDURE VER_VENTA (ID NUMBER) AS
  IMPORTE NUMBER;
  TOTAL_V NUMBER;
  CLI TIP_CLIENTE := TIP_CLIENTE(NULL, NULL, NULL, NULL, NULL);
  FEC DATE;
--cursor para recorrer la tabla anidada del idventa
--que se recibe, recorre las líneas de venta
  CURSOR C1 IS
```

```
  SELECT NUMEROLINEA LIN, DEREF(IDPRODUCTO) PROD, CANTIDAD
  FROM THE
    (SELECT T.LINEAS FROM TABLA_VENTAS T WHERE IDVENTA = ID);

BEGIN
--obtener datos de la venta
  SELECT DEREF(IDCLIENTE), FECHAVENTA, V.TOTAL_VENTA()
         INTO CLI, FEC, TOTAL_V
  FROM TABLA_VENTAS V WHERE IDVENTA = ID;

  DBMS_OUTPUT.PUT_LINE('NÚMERO DE VENTA: ' || ID ||
                       ' * Fecha de venta: ' || FEC);

  DBMS_OUTPUT.PUT_LINE('CLIENTE: ' || CLI.NOMBRE);
  DBMS_OUTPUT.PUT_LINE('DIRECCION: '|| CLI.DIREC.CALLE);
  DBMS_OUTPUT.PUT_LINE('==========================================');

  FOR I IN C1 LOOP
    IMPORTE:= I.CANTIDAD * I.PROD.PVP;
    DBMS_OUTPUT.PUT_LINE(I.LIN|| '*' || I.PROD.DESCRIPCION || '*' ||
                I.PROD.PVP || '*' || I.CANTIDAD || '*' ||IMPORTE);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total Venta: ' || TOTAL_V);
END VER_VENTA;
/

--Ejecutamos el procedimiento para visualizar los datos de la venta 2:

BEGIN
  VER_VENTA(2);
END;
/

NÚMERO DE VENTA: 2 * Fecha de venta: 10/03/16
CLIENTE: Luis Gracia
DIRECCION: C/Las Flores 23
==========================================
1*CAJA DE CRISTAL DE MURANO*100*2*200
2*OPERACIONES CON BD*25*1*25
3*APLICACIONES WEB*25,5*4*102
Total Venta: 327
```

ACTIVIDAD 4.9

Crea una función almacenada que reciba un identificador de venta y retorne el total de la venta. Comprueba si la venta existe, si no existe devuelve -1. Realiza un bloque PL/SQL anónimo que haga uso de la función.

Realiza una consulta que muestre por cada producto el total de unidades vendidas, debe mostrar el identificador del producto, la descripción y la suma de las unidades vendidas.

4.3. BASES DE DATOS ORIENTADA A OBJETOS

Las **Bases de Datos Orientadas a Objetos (BDOO)** son aquellas cuyo modelo de datos está orientado a objetos, soportan el paradigma orientado a objetos almacenando métodos y datos. Su origen se debe principalmente a la existencia de problemas para representar cierta información y modelar ciertos aspectos del mundo real. Las BDOO simplifican la programación orientada a objetos (POO) almacenando directamente los objetos en la BD y empleando las mismas estructuras y relaciones que los lenguajes de POO.

Podemos decir que un **Sistema Gestor de Base de Datos Orientada a Objetos (SGBDOO)** es un sistema gestor de base de datos (SGBD) que almacena objetos.

4.3.1. Características de las bases de datos OO

Las características asociadas a las BDOO son las siguientes:

- Los datos se almacenan como **objetos**.
- Cada objeto se identifica mediante un identificador único u **OID (*Object Identifier*)**, este identificador no es modificable por el usuario.
- Cada objeto define sus métodos y atributos y la interfaz mediante la cual se puede acceder a él, el usuario puede especificar qué atributos y métodos pueden ser usados desde fuera.
- En definitiva, un SGBDOO debe cumplir las características de un SGBD: **persistencia, concurrencia, recuperación ante fallos, gestión del almacenamiento secundario y facilidad de consultas**; y las características de un sistema orientado a objetos (OO): **encapsulación, identidad, herencia y polimorfismo**.

En 1989 se hizo el manifiesto *Malcolm Atkinson* que propone 13 características obligatorias para los SGBDOO basado en dos criterios: debe ser un sistema orientado a objetos y debe ser un SGBD. Las características son:

1. Debe soportar objetos complejos.
2. Identidad del objeto: todos los objetos deben tener un identificador que sea independiente de los valores de sus atributos.
3. Encapsulamiento: los programadores solo tendrán acceso a la interfaz de los métodos, de modo que sus datos e implementación estén ocultos.
4. Soporte para tipos o clases.
5. Herencia: un subtipo o una subclase heredará los atributos y métodos de su supertipo o superclase, respectivamente.
6. Debe soportar sobrecarga: los métodos deben poder aplicarse a diferentes tipos.
7. El DML debe ser completo.
8. El conjunto de tipos de datos debe ser extensible.
9. Debe soportar persistencia de datos: los datos deben mantenerse después de que la aplicación que los creó haya finalizado.

10. Debe ser capaz de manejar grandes BD: debe proporcionar mecanismos que aseguren independencia entre los niveles lógico y físico del sistema.
11. Debe soportar concurrencia.
12. Debe ser capaz de recuperarse ante fallos hardware y software.
13. Debe proporcionar un método de consulta sencillo.

¡¡INTERESANTE!!

The Object-Oriented Database System Manifesto: En este trabajo se intenta definir un sistema de base de datos orientada a objetos. En él se describen los principales rasgos y características que un sistema debe tener para calificarle como un sistema de base de datos orientado a objetos:

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

Las **ventajas** que aporta un SGBDOO son las siguientes:

- Mayor capacidad de modelado. La utilización de objetos permite representar de una forma más natural los datos que se necesitan guardar.
- Extensibilidad. Se pueden construir nuevos tipos de datos a partir de tipos existentes.
- Existe una única interfaz entre el LMD (lenguaje de manipulación de datos) y el lenguaje de programación. Esto elimina el tener que incrustar un lenguaje declarativo como SQL en un lenguaje imperativo como Java o C.
- Lenguaje de consultas más expresivo. El lenguaje de consultas es navegacional de un objeto al siguiente, en contraste con el lenguaje declarativo SQL.
- Soporte a transacciones largas, necesario para muchas aplicaciones de bases de datos avanzadas.
- Adecuación a aplicaciones avanzadas de bases de datos (CASE, CAD, sistemas multimedia, etc.)

Entre los **inconvenientes** hay que destacar:

- Falta de un modelo de datos universal, la mayoría de los modelos carecen de una base teórica.
- Falta de experiencia, el uso de los SGBDOO es todavía relativamente limitado.
- Falta de estándares, no existe un lenguaje de consultas estándar como SQL, aunque está el lenguaje **OQL** (***Object Query Language***) de **ODMG** que se está convirtiendo en un estándar de facto.
- Competencia con los SGBDR y los SGBDOR, que tienen gran experiencia de uso.
- La optimización de consultas compromete la encapsulación: optimizar consultas requiere conocer la implementación para acceder a la BD de una manera eficiente.
- Complejidad: el incremento de funcionalidad provisto por un SGBDOO lo hace más complejo que un SGBDR. La complejidad conlleva productos más caros y difíciles de usar.
- Falta de soporte a las vistas: la mayoría de SGBDOO no proveen mecanismos de vistas.
- Falta de soporte a la seguridad.

4.3.2. El estándar ODMG

ODMG (***Object Database Management Group***) es un grupo formado por fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. Uno de sus estándares, el cual lleva el mismo nombre del grupo (**ODMG**) especifica los elementos que se definirán, y en qué manera se hará, para la consecución de persistencia en las BDOO que soporten el estándar.

La última versión del estándar, **ODMG 3.0** propone los siguientes componentes:

- Modelo de objetos.
- Lenguaje de definición de objetos (**ODL**, *Object Definition Language*).
- Lenguaje de consulta de objetos (**OQL**, *Object Query Language*).
- Conexión con los lenguajes C++, Smalltalk y Java.

El modelo de objetos ODMG especifica las características de los objetos, cómo se relacionan, cómo se identifican, construcciones soportadas, etc. Las primitivas de modelado básicas son: los objetos caracterizados por un identificador único (**OID**) y los literales que son objetos que no tienen identificador, no pueden aparecer como objetos, están embebidos en ellos.

Los tipos de objetos son:

- **Atómicos**: boolean, *short, long, unsigned long, unsigned short, float, double, char, string, enum, octect.*
- **Tipos estructurados**: *date, time, timestamp, interval.*
- **Colecciones** *<interfaceCollection>*:

- *set<tipo>*: grupo desordenado de objetos del mismo tipo que no admite duplicados.
- *bag<tipo>*: grupo desordenado de objetos del mismo tipo que permite duplicados.
- *list<tipo>*: grupo ordenado de objetos del mismo tipo que permite duplicados.
- *array<tipo>*: grupo ordenado de objetos del mismo tipo a los que se puede acceder por su posición. El tamaño es dinámico.
- *dictionary<clave,valor>*: grupo de objetos del mismo tipo, cada valor está asociado a su clave.

Los **literales** pueden ser atómicos (long, short, boolean, unsigned long, etc.), colecciones (set, bag, list, array, dictionary), estructuras (date, interval, time, timestamp) y NULL.

Mediante las **Clases** especificamos el estado y el comportamiento de un tipo de objeto, pueden incluir métodos. Son instanciables, por lo que a partir de ellas se pueden crear instancias de objetos individuales. Son equivalentes a una clase concreta en los lenguajes de programación. Una clase es un tipo de objeto asociado a un "**extent**".

El lenguaje **ODL** es el equivalente al lenguaje de definición de datos (DDL) de los SGBD tradicionales. Define los atributos, las relaciones entre los tipos y especifica la signatura de las operaciones. La sintaxis de ODL extiende el lenguaje de definición de interfaces de CORBA (*Common Object Request Broker Architecture*). Algunas de las palabras reservadas para definir los objetos son:

- ***class***: declaración del objeto, define el comportamiento y el estado de un tipo de objeto.

- ***extent***: define la extensión, nombre para el actual conjunto de objetos de la clase. En las consultas se hace referencia al nombre definido en esta cláusula, no se hace referencia al nombre definido a la derecha de ***class***.
- ***key[s]***: declara la lista de claves para identificar las instancias.
- ***attribute***: declara un atributo.
- ***set*** | ***list*** | ***bag*** | ***array***: declara un tipo de colección.
- ***struct***: declara un tipo estructurado.
- ***enum***: declara un tipo enumerado.
- ***relationship***: declara una relación.
- ***inverse***: declara una relación inversa.
- ***extends***: define la herencia simple.

Veamos como se puede definir un objeto Cliente, similar al tipo cliente visto anteriormente, la clave es el NIF. Se definen los atributos y un método; uno de los atributos es un tipo estructurado (*struct*), otro es enumerado (*enum*) y también tenemos un tipo colección (*set*):

```
class Cliente (extent Clientes key NIF)
{
  /*Definición de atributos*/
    attribute struct Nombre_Persona {
                     string apellidos,
                     string nombrepern} nombre;
    attribute string NIF;
    attribute date fecha_nac;
    attribute enum Genero{H,M} sexo;
    attribute struct Direccion{
                     string calle,
                     string poblac} direc;
    attribute set<string> telefonos;

  /*Definición de operaciones*/
    short edad();
}
```

Definimos el objeto Producto:

```
class Producto (extent Productos key IDPRODUCTO)
{
  /*Definición de atributos*/
    attribute short IDPRODUCTO;
    attribute string descripcion;
    attribute float pvp;
    attribute short stockactual;
}
```

Definimos el objeto Línea de Venta con datos de la línea y la operación para calcular el importe de la línea:

```
class LineaVenta (extent Lineaventas)
{
```

```
  /*Definición de atributos*/
    attribute short numerolinea;
    attribute Producto product;
    attribute short cantidad;

  /*Definición de operaciones*/
    float importe();
}
```

A continuación definimos el objeto Venta y sus relaciones: una venta pertenece a un cliente (*pertenece_a_cliente*) y la inversa, el cliente tiene venta (*tiene_venta*), también se define un atributo colección (*set*) para las líneas de venta:

```
class Venta (extent Ventas key IDVENTA)
{
  /*Definición de atributos*/
    attribute short IDVENTA;
    attribute date fecha_venta;
    attribute set <LineaVenta> lineas;

  /*Definición de relaciones*/
    relationship Cliente pertenece_a_cliente inverse
                 Cliente::tiene_venta;

  /*Definición de operaciones*/
    float total_venta();
}
```

4.3.3. El lenguaje de consultas OQL

OQL (***Object Query Language***) es el lenguaje estándar de consultas de BDOO. Las características son las siguientes:

- Es orientado a objetos y está basado en el modelo de objetos de la **ODMG**.
- Es un lenguaje declarativo del tipo de SQL. Su sintaxis básica es similar a SQL.
- Acceso declarativo a los objetos de la base de datos (propiedades y métodos).
- Semántica formal bien definida.
- No incluye operaciones de actualización (solo de consulta). Las modificaciones se realizan mediante los métodos que los objetos poseen.
- Dispone de operadores sobre colecciones (*max, min, count*, etc.) y cuantificadores (*for all, exists*).

La sintaxis básica de OQL es una estructura SELECT como en SQL:

```
SELECT <lista de valores>
FROM <lista de colecciones y miembros típicos>
[WHERE <condición>]
```

Donde las colecciones en FROM pueden ser extensiones (los nombres que aparecen a la derecha de **extent**) o expresiones que evalúan una colección. Se suele utilizar una variable

iterador que vaya tomando valores de los objetos de la colección. Las variables iterador se pueden especificar de varias formas utilizando las cláusulas IN o AS:

```
FROM Clientes x
FROM x IN Clientes
FROM Clientes AS x
```

Para acceder a los atributos y objetos relacionados se utilizan expresiones de camino. Una expresión de camino empieza normalmente con un nombre de objeto o una variable iterador seguida de atributos conectados mediante un punto o nombres de relaciones. Por ejemplo, para obtener el nombre de los clientes que son mujeres, podemos escribir:

```
SELECT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo = "M"
SELECT x.nombre.nombreper FROM Clientes x WHERE x.sexo = "M"
SELECT x.nombre.nombreper FROM Clientes AS x WHERE x.sexo = "M"
```

En general, supongamos que *v* es una variable cuyo tipo es Venta:

- *v.IDVENTA* es el identificador de venta del objeto *v*.
- *v.fecha_venta* es la fecha de venta del objeto *v*.
- *v.total_venta()* obtiene el total venta del objeto *v*.
- *v.pertenece_a_cliente* es un puntero al cliente mencionado en *v*.
- *v.pertenece_a_cliente.direc* es la dirección del cliente mencionado en *v*.
- *v.lineas* es una colección de objetos del tipo LineaVenta.
- El uso de *v.lineas.numerolinea* NO es correcto porque *v.lineas* es una colección de objetos y no un objeto simple.
- Cuando tenemos una colección como en *v.lineas*, para poder acceder a los atributos de la colección podemos usar la orden FROM.

Ejemplos:

- Obtener los datos del cliente cuyo IDVENTA = 1:

```
SELECT v.pertenece_a_cliente.nombre, v.pertenece_a_cliente.direc,
       v.fecha_venta, v.total_venta()
FROM Ventas v WHERE v.IDVENTA = 1;
```

- Obtenemos las líneas de venta del IDVENTA = 1, en este ejemplo el objeto v es usado para definir la segunda colección *v.lineas*:

```
SELECT lin.numerolinea, lin.product.descripcion,
       lin.cantidad, lin.importe()
FROM Ventas v, v.lineas lin WHERE v.IDVENTA = 1;
```

El resultado de una consulta **OQL** puede ser de cualquier tipo soportado por el modelo. Por ejemplo, la consulta anterior devuelve un conjunto de estructuras del tipo *short, string,* y *float*; el resultado es del tipo colección: *bag (struct(numerolinea:short, descripcion:string, cantidad: short, importe: float)).*

En cambio la consulta: *SELECT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo = "M"*, devuelve un conjunto de nombres; el tipo devuelto es: *bag(string)*.

Recordemos la diferencia entre las colecciones *set* y *bag*, *set* es un grupo desordenado de objetos del mismo tipo que no permite duplicados y *bag* permite duplicados.

Se pueden usar alias en las consultas, por ejemplo, la SELECT anterior: *SELECT lin.numerolinea, lin.product.descripcion, lin.cantidad, lin.importe()*; se puede expresar usando alias de la siguiente manera:

```
SELECT nlin:lin.numerolinea,
       dpro:lin.product.descripcion,
       can:lin.cantidad,
       imp:lin.importe()
```

Y el tipo devuelto en este caso es: *bag (struct(nlin:short, dpro:string, can: short, imp: float))*.

Para obtener un set de estructuras (colección que no admite duplicados) podemos usar DISTINCT a la derecha de SELECT:

```
SELECT DISTINCT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo ="M"
```

En este caso el tipo devuelto es: *set(string)*.

Para obtener una lista (un tipo *list*) de estructuras usamos la cláusula ORDER BY:

```
SELECT nlin:lin.numerolinea,
       dpro:lin.product.descripcion,
       can:lin.cantidad,
       imp:lin.importe()
FROM Ventas v, v.lineas lin WHERE v.IDVENTA = 1 ORDER BY nlin ASC;
```

Y el tipo devuelto en este caso es: *list(struct(nlin:short, dpro:string, can: short, imp: float))*.

4.3.3.1. Operadores de comparación

Los valores pueden ser comparados usando los siguientes operadores:

= Igual a

> Mayor que

>= Mayor o igual que

< Menor que

<= Menor o igual que

!= Distinto de

Para comparar cadenas de caracteres podemos usar los operadores **IN** y **LIKE**:

- **IN**: comprueba si existe un carácter en una cadena de caracteres: *carácter IN cadena*.
- **LIKE**: comprueba si dos cadenas son iguales: *cadena1 LIKE cadena2*. *cadena2* puede contener caracteres especiales:

  _ o ?: indicador de posición, representa cualquier carácter.

*o %: representa una cadena de caracteres.

Ejemplos:

- Obtener los datos de las ventas de los clientes de la población de TOLEDO y cuyos apellidos empiecen por la letra A:

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta()
FROM Ventas v
WHERE v.pertenece_a_cliente.direc.pobla = "TOLEDO"
AND   v.pertenece_a_cliente.nombre.apellidos LIKE "A%";
```

- Obtener para el IDVENTA 1 aquellas líneas de venta cuya descripción del producto contenga el carácter P en su descripción:

```
SELECT lin.numerolinea, lin.product.descripcion,
       lin.cantidad, lin.importe()
FROM Ventas v, v.lineas lin
WHERE v.IDVENTA = 1 AND 'P' IN lin.product.descripcion;
```

4.3.3.2. Cuantificadores y operadores sobre colecciones

Mediante el uso de cuantificadores podemos comprobar si todos los miembros, al menos un miembro, o algunos miembros, etc. satisfacen alguna condición:

Todos los miembros: *FOR ALL x IN colección : condición*

Al menos uno: *EXISTS x IN colección : condición*

EXISTS x

Solo uno: *UNIQUE x*

Algunos / cualquier: *colección comparación SOME/ANY condición*

Donde *comparación* puede ser : <, >, <=, >=, o =

Ejemplos:

- Obtener todas las ventas que tengan líneas de venta cuya descripción del producto sea "PNY Pendrive 16 GB":

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta() FROM Ventas v
WHERE EXISTS x IN
v.lineas : x.product.descripcion = "PNY Pendrive 16 GB";
```

- Obtener las ventas que solo tienen líneas de venta cuya descripción de producto es "PNY Pendrive 16 GB":

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta() FROM Ventas v
WHERE FOR ALL x IN
            v.lineas : x.product.descripcion = "PNY Pendrive 16 GB";
```

Los operadores AVG, SUM, MIN, MAX y COUNT, se pueden aplicar a cualquier colección, siempre y cuando tengan sentido para el tipo de elemento. Por ejemplo, para calcular la media del total venta de todas las ventas necesitaríamos asignar el valor devuelto a una variable:

```
Media = AVG( SELECT v.total_venta() FROM Ventas v )
```

El tipo devuelto es una colección de un elemento: *bag(struct(total_venta: float))*.

Como hemos visto **OQL** es bastante complejo y actualmente ningún creador de software lo ha implementado completamente. **OQL** ha influenciado el diseño de algunos lenguajes de consulta nuevos como JDOQL (*Java Data Objects Query Language*) y EJBQL (*Enterprise Java Bean Query Language*), pero estos no pueden ser considerados como versiones de **OQL**.

4.4. EJEMPLO DE BDOO

En el capítulo 3 se presentó la base de datos orientada a objetos **DB4o**, a continuación se presenta una base de datos sencilla de utilizar que aporta una API simple que no requiere el aprendizaje de técnicas de mapeo, se trata de **NeoDatis Object Database**. *NeoDatis ODB* es una base de datos orientada a objetos de código abierto que funciona con Java, .Net, Groovy y Android. Los objetos se pueden almacenar y recuperar con una sola línea de código, sin necesidad de tener que mapearlos a tablas.

Desde la web *http://neodatis.wikidot.com/downloads* podemos descargar la última versión. Para los ejemplos se ha descargado el fichero **neodatis-odb-1.9.30.689.zip**. Lo descomprimimos y copiamos el fichero **neodatis-odb-1.9.30.689.jar** en la carpeta adecuada para después definirlo en el CLASSPATH o incluirlo en nuestro proyecto Eclipse o NetBeans. Desde la carpeta */doc/javadoc* podemos acceder a la documentación de la API de *Neodatis ODB*.

El ejemplo que se presenta a continuación almacena objetos Jugadores en la base de datos de nombre *neodatis.test*. Para abrir la base de datos se usa la clase **ODBFactory** con el método ***open()*** que devuelve un **ODB** (es la interfaz principal, lo que el usuario ve):

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
```

Para almacenar los objetos se usa el método ***store()***:

```
Jugadores j4 = new Jugadores("Alicia", "tenis", "Madrid",14);
odb.store(j4);
```

Una vez almacenados los objetos, para recuperarlos usamos el método ***getObjects()***, que recibe la clase cuyos objetos se van a recuperar y devuelve una colección de objetos de esa clase:

```
Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
```

Para validar los cambios en la base de datos se usa el método ***close()***. El código completo del ejemplo (*EjemploNeodatis.java*) donde se ha incluido la clase Jugadores es el siguiente:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
//Clase Jugadores
class Jugadores {
    private String nombre;
    private String deporte;
    private String ciudad;
    private int edad;
```

```
    public Jugadores() {}
    public Jugadores(String nombre, String deporte,
                     String ciudad, int edad) {
       this.nombre = nombre;
       this.deporte = deporte;
       this.ciudad = ciudad;
       this.edad = edad;
    }

    public void setNombre(String nombre) {this.nombre = nombre;}
    public String getNombre() {return nombre;}
    public void setDeporte(String deporte) {this.deporte = deporte;}
    public String getDeporte() {return deporte;}
    public void setCiudad(String ciudad) {this.ciudad = ciudad;}
    public String getCiudad () {return ciudad;}
    public void setEdad(int edad) {this.edad = edad;}
    public int getEdad() {return edad;}
}
//
public class EjemploNeodatis {
    public static void main(String[] args) {

      // Crear instancias para almacenar en BD
      Jugadores j1 = new Jugadores("Maria", "voleibol", "Madrid", 14);
      Jugadores j2 = new Jugadores("Miguel", "tenis", "Madrid", 15);
      Jugadores j3 = new Jugadores
                     ("Mario", "baloncesto", "Guadalajara", 15);
      Jugadores j4 = new Jugadores("Alicia", "tenis", "Madrid", 14);

      ODB odb = ODBFactory.open("neodatis.test");// Abrir BD

      // Almacenamos objetos
      odb.store(j1);
      odb.store(j2);
      odb.store(j3);
      odb.store(j4);

      //recuperamos todos los objetos
      Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
      System.out.printf("%d Jugadores: %n", objects.size());

      int i = 1;
      // visualizar los objetos
      while(objects.hasNext()){
         Jugadores jug = objects.next();
         System.out.printf("%d: %s, %s, %s %n",
                     i++, jug.getNombre(), jug.getDeporte(),
                     jug.getCiudad(), jug.getEdad());
      }
    odb.close(); // Cerrar BD
  }
}
```

ACTIVIDAD 4.10

Crea un nuevo proyecto en Eclipse y añade el JAR para trabajar con Neodatis. Dentro del proyecto crea un paquete de nombre clases. Crea la clase Paises con dos atributos y sus *getter* y *setter*. Los atributos son: `private int id; private String nombrepais;`

Añade también el método *toString()* para que devuelva el nombre del país: `public String toString(){return nombrepais;}`

Crea la clase Jugadores (en el paquete clases, como en el ejemplo anterior) y añade el siguiente atributo con sus *getter* y *setter*: `private Paises pais;`

Crea una clase Java (con el método *main()*) que cree una base de datos de nombre EQUIPOS.DB e inserte países y los jugadores de esos países. Añade otra clase Java para visualizar los países y los jugadores que hay en la BD.

NeoDatis dispone de un explorador que nos permite navegar por los objetos. Para ejecutarlo hacemos doble clic en el fichero **odb-explorer.bat** (sistemas Windows) u **odb-explorer.sh** (sistemas Linux). El explorador también se puede abrir haciendo doble clic en el fichero **neodatis-odb-1.9.30.689.jar**.

En primer lugar es necesario abrir la base de datos por la que vamos a navegar, pulsamos en el menú ***NeoDatis ODB->Open Database*** (Figura 4.2).

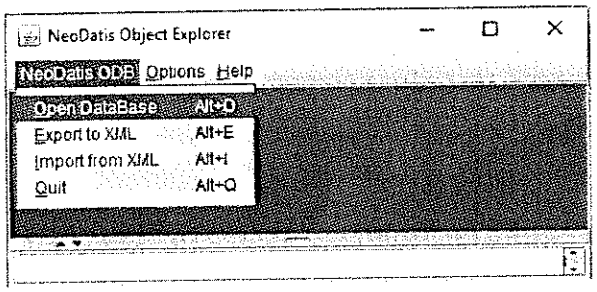


Figura 4.2. NeoDatis Open DataBase.

Se abre una nueva ventana con 2 pestañas, nos quedamos en la primera (*Local connections*), localizamos en nuestro disco el fichero *neodatis.test* y pulsamos el botón *Connect*. Se abre el explorador, al pulsar con el botón derecho del ratón sobre la clase *Jugadores* podemos acceder a la vista de los objetos, vista en formato de tabla, realizar consultas, crear un nuevo objeto, etc.; véase Figura 4.3. Desde el explorador se pueden modificar los objetos, eliminarlos, etc. Después de realizar cada operación hemos de pulsar el botón *Commit* para validar los cambios. Para finalizar con la base de datos pulsamos el botón *Close Database*.

Podemos acceder a los objetos conociendo su **OID** (identificador de objeto). El siguiente ejemplo muestra los datos del objeto cuyo **OID** es el 3:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.OID;
import org.neodatis.odb.core.oid.OIDFactory;

public class ejemploOid {
  public static void main(String[] args) {
    ODB odb = ODBFactory.open("neodatis.test"); // Abrir BD
    OID oid = OIDFactory.buildObjectOID(3); //Obtener objeto con OID 3
    Jugadores jug  = (Jugadores) odb.getObjectFromId(oid);
    System.out.printf("%s, %s, %s, %d %n", jug.getNombre(),
```

```
                    jug.getDeporte(), jug.getCiudad(), jug.getEdad());
        odb.close(); // Cerrar BD
    }
}
```

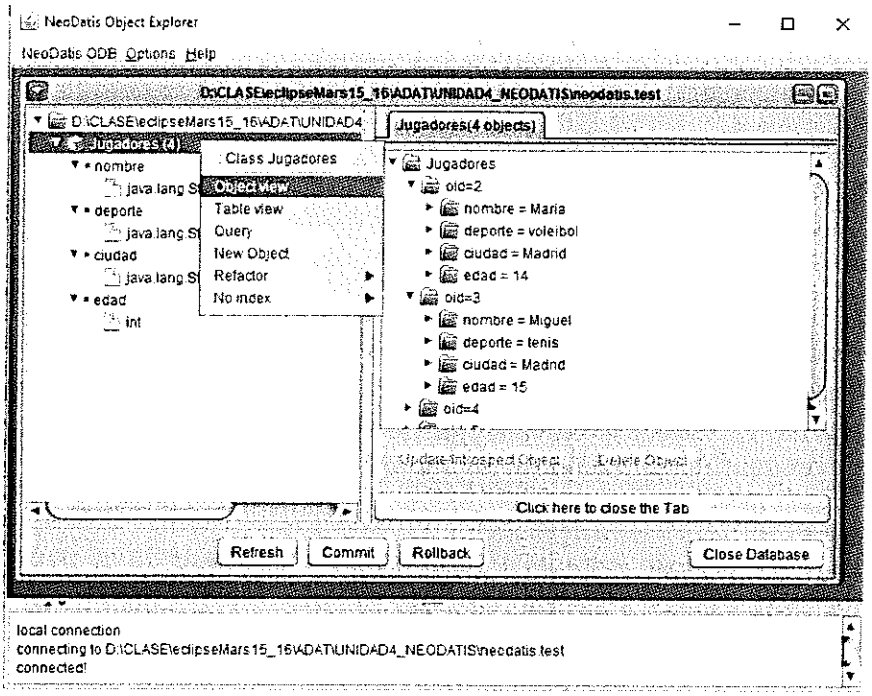


Figura 4.3. Explorador de objetos de NeoDatis.

El **OID** de un objeto es devuelto por los métodos ***store(Objeto)*** y ***getObjectId(Objeto)***, se necesita importar el paquete *import org.neodatis.odb.OID*. Por ejemplo, para obtener el **OID** del objeto *j1* podemos hacerlo de las siguientes maneras:

```
OID oid1 = odb.store(j1);
OID oid1 = odb.getObjectId(j1) ;
```

4.4.1. Consultas sencillas

Para realizar consultas usamos la clase **CriteriaQuery** en donde especificaremos la clase y el criterio para realizar la consulta. Necesitaremos importar los siguientes paquetes:

```
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
```

El siguiente ejemplo obtiene todos los jugadores que practican el deporte tenis:

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
IQuery query = new
       CriteriaQuery(Jugadores.class, Where.equal("deporte", "tenis"));

query.orderByAsc("nombre, edad"); //Ordena ascendentemente
                                  //por nombre y edad
Objects<Jugadores> objects = odb.getObjects(query); //Obtiene todos los
                                                    //jugadores
```

Para obtener el primer objeto usamos el método ***getFirst()***:

```
//obtiene solo el 1°
Jugadores j = (Jugadores) odb.getObjects(query).getFirst();
```

Este método lanza la excepción **IndexOutOfBoundsException** si no localiza ningún objeto con ese criterio. Para capturarla pondríamos lo siguiente:

```
try{
    Jugadores j = (Jugadores) odb.getObjects(query).getFirst();
}catch(IndexOutOfBoundsException e){
    System.out.printf("OBJETO NO LOCALIZADO");
 }
```

Como se puede ver el método ***CriteriaQuery()*** utiliza ***Where.equal*** para seleccionar los objetos que cumplan la condición especificada. Para ordenar la salida ascendentemente se usa el método ***orderByAsc()***, entre paréntesis se ponen los atributos por los que se realiza la ordenación; para ordenar descendentemente se usa ***orderByDesc()***.

ACTIVIDAD 4.11

Añade al proyecto Eclipse creado anteriormente otra clase Java para realizar la consulta anterior, pero por cada jugador visualiza también su país.

Para modificar un objeto, primero es necesario cargarlo, después lo modificamos usando los métodos *set* del objeto y a continuación lo actualizamos con el método ***store()***. Los cambios que se realicen en la base de datos se validarán utilizando el método ***commit()***, aunque también se validarán cerrando la base de datos con el método ***close()***.

El siguiente ejemplo cambia el deporte de *María* a *vóley-playa*:

```
IQuery query = new
       CriteriaQuery(Jugadores.class,Where.equal("nombre", "Maria"));
Objects<Jugadores> objetos = odb.getObjects(query);

// Obtiene solo el primer objeto encontrado
Jugadores jug = (Jugadores) objetos.getFirst();
jug.setDeporte("vóley-playa"); // Cambia el deporte
odb.store(jug);  // Actualiza el objeto
odb.commit();    // Valida los cambios
```

Para eliminar un objeto, primero lo localizamos como antes y luego usamos el método ***delete()***:

```
odb.delete(jug); // elimina el objeto
```

Con **CriteriaQuery** se puede usar la interfaz **ICriterion** para construir el criterio de la consulta, para usarlo será necesario importar otros paquetes:

```
import org.neodatis.odb.core.query.criteria.ICriterion;
```

Por ejemplo, para obtener los jugadores cuya edad es 14 utilizamos el criterio ***Where.equal()***:

```
ICriterion criterio = Where.equal("edad", 14);
CriteriaQuery query = new CriteriaQuery(Jugadores.class, criterio);
```

```
Objects<Jugadores> objects = odb.getObjects(query);
```

Para obtener los jugadores cuyo nombre empieza por la letra M usamos ***Where.like()***:

```
ICriterion criterio = Where.like("nombre", "M%");
```

Para obtener los jugadores cuya edad es > que 14 usamos ***Where.gt()***:

```
ICriterion criterio = Where.gt("edad", 14);
```

Para mayor o igual que 14 usamos: ***Where.ge("edad", 14).***

Para menor que 14 usamos: ***Where.lt("edad", 14).***

Para menor o igual que 14 usamos: ***Where.le("edad", 14).***

Para comprobar si un array o una colección contiene un valor determinado usamos ***Where.contain()***:

```
ICriterion criterio = Where.contain("nombrearray", valor);
```

Para comprobar si un atributo es nulo usamos ***Where.isNull()***:

```
ICriterion criterio = Where.isNull("atributo");
```

Para comprobar si no es nulo usamos ***Where.isNotNull()***:

```
ICriterion criterio = Where.isNotNull("atributo");
```

La construcción de expresiones lógicas añade complejidad al criterio de consulta. Es necesario importar los paquetes:

```
import org.neodatis.odb.core.query.criteria.And;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Not;
```

Y añadir mediante el método ***add()*** a los criterios de búsqueda. Por ejemplo, para obtener los jugadores de Madrid y edad 15 escribimos el siguiente criterio **AND**:

```
ICriterion criterio = new And().add(Where.equal("ciudad", "Madrid"))
                              .add(Where.equal("edad", 15));
```

Para obtener los jugadores cuya ciudad sea Madrid o la edad sea >= que 15 construimos el criterio **OR**:

```
ICriterion criterio = new Or().add(Where.equal("ciudad", "Madrid"))
                             .add(Where.ge("edad", 15));
```

Para obtener los jugadores cuyo nombre no empiece por la letra M usamos ***Where.not()***:

```
ICriterion criterio = Where.not(Where.like("nombre", "M%"));
```

O también se puede poner:

```
ICriterion criterio1 = Where.like("nombre", "M%");
```

```
ICriterion criterio = Where.not(criterio1);
```

Consulta la página *http://neodatis-odb.wikidot.com/criteria-queries* para saber más sobre la API **CriteriaQuery**.

Ejemplos:

- El siguiente método visualiza los jugadores de 14 años de los países de IRLANDA, FRANCIA e ITALIA.

```
private static void jugadores14irlandafranciaitalia() {
  ODB odb = ODBFactory.open("EQUIPOS.DB");
  ICriterion criterio = new And().add(Where.equal("edad", 14))
      .add(new Or().add(Where.equal("pais.nombrepais", "IRLANDA"))
                   .add(Where.equal("pais.nombrepais", "ITALIA"))
                   .add(Where.equal("pais.nombrepais", "FRANCIA")));

  IQuery query = new CriteriaQuery(Jugadores.class, criterio);
  Objects jugadores = odb.getObjects(query);

  if (jugadores.size() == 0) {
      System.out.println(" No existen jugadores de 14 años de  IRLANDA,
                          ITALIA, FRANCIA.");
  } else {
      Jugadores jug;
      System.out.println("Jugadores de 14 años de IRLANDA, ITALIA,
                         FRANCIA.");
      while (jugadores.hasNext()) {
       jug = (Jugadores) jugadores.next();
       System.out.printf("Nombre: %s, Edad: %d, Ciudad: %s,Pais: %s%n",
                       jug.getNombre(), jug.getEdad(),
                       jug.getCiudad(), jug.getPais().getNombrepais());
      }
  }
  odb.close();
}
```

- El siguiente método recibe el nombre de un país y actualiza las edades de los jugadores de ese país. Suma 2 a la edad. Si no hay jugadores del país visualiza un mensaje indicándolo:

```
private static void actualizaredadjugadoresdepais(String pais) {
  ODB odb = ODBFactory.open("EQUIPOS.DB");
  ICriterion crit = Where.equal("pais.nombrepais", pais);
  IQuery query = new CriteriaQuery(Jugadores.class, crit);

  Objects jugadores = odb.getObjects(query);

  if (jugadores.size() == 0) {
      System.out.printf(" No existen Jugadores de %s,
                          no se actualiza la edad %n", pais);
  } else {
     Jugadores jug;
     System.out.printf("ACTUALIZAMOS LA EDAD DE LOS JUGADORES DE %s%n",
                       pais);
```

```
    odb.close();
}
```

ACTIVIDAD 4.13

Realiza un método que reciba un nombre de país y visualice el número de jugadores que tiene por cada ciudad de ese país, y la media de edad. Si no tiene jugadores que visualice que el país no tiene jugadores. Y si el país no existe que visualice que el país no existe. Evita el error del redondeado de la media.

4.4.2.1 Crear una BD Neodatis a partir de un modelo relacional

En este apartado vamos a ver como crear una BD Neodatis, a partir de los datos de las tablas de un modelo relacional. Disponemos de las siguientes tablas en MySQL:

- ***C1_Centros***: con información de los centros de una red de centros. Cada centro tiene un profesor que es el director del centro. Un centro tiene muchos profesores.
- ***C1_Profesores***: con información de los profesores de los centros. El profesor pertenece a un centro. Los profesores imparten asignaturas.
- ***C1_Asigprof:*** con información de los profesores y las asignaturas que imparten. Un profesor imparte varias asignaturas. Una asignatura puede ser impartida por varios profesores.
- ***C1_Asignaturas:*** con información de las asignaturas y su nombre.El modelo de datos y las relaciones se muestran en la figura:

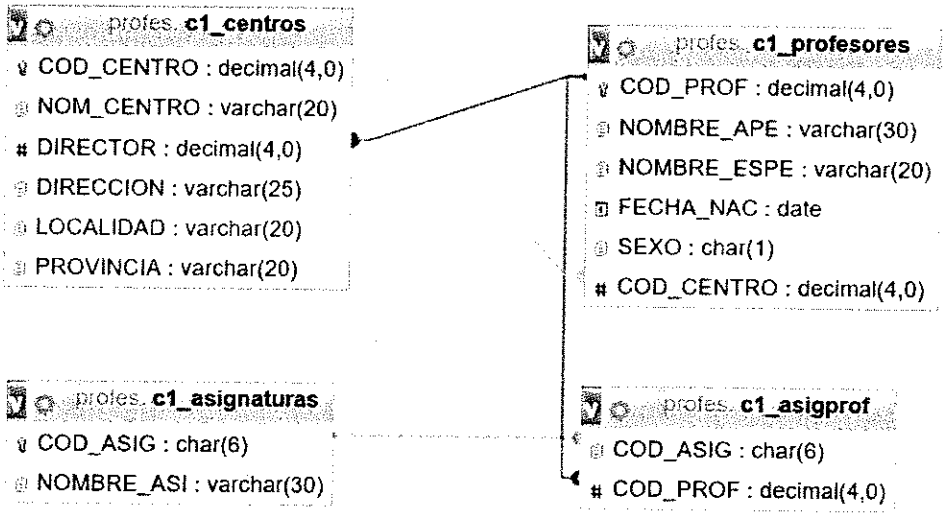


Figura 4.4. Modelo de datos BD MySql, profesores.

Deseamos crear la BD Neodatis *profesasig.neo* con las siguientes clases:

```
public class C1Asignaturas {
    private String codAsig;
    private String nombreAsi;
    private Set<C1Profesores> setprofesores;
. . . . . . . . . . . . . . }

public class C1Centros
      private Integer codCentro;
      private String nomCentro;
```

```
      private C1Profesores director;
      private String direccion;
      private String localidad;
      private String provincia;
      private Set<C1Profesores> setprofesores;
. . . . . . . . . . . . . . }
public class C1Profesores {
      private Integer codProf;
      private String nombreApe;
      private String nombreEspe;
      private Date fechaNac;
      private String sexo;
      private C1Centros c1Centros;
. . . . . . . . . . . . . . }
```

- La clase ***C1Asignaturas*** contendrá un set con los profesores que imparten esa asignatura.
- La clase ***C1Centros*** contendrá un set con los profesores de ese centro y un objeto profesor que es el director.
- La clase ***C1Profesores*** contendrá un objeto con los datos de su centro.

Para la solución se hace lo siguiente:

Se crea el atributo de clase ***static ODB bd.***

Y desde el método *main()*, creamos la conexión con MySQL, abrimos la BD Neodatis y llamaremos a los distintos métodos para crear las clases de Neodatis:

```
public static void main(String[] args) {
try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection conexion = DriverManager.getConnection
          ("jdbc:mysql://localhost/profes", "root", "");
      bd = ODBFactory.open("profesasig.neo");
      // Recorrer C1asignaturas y guardar en Neodatis
      InsertarAsignaturas(conexion);
      // Recorrer C1Centros y guardar en Neodatis
      InsertarCentros(conexion);
      // Recorrer C1Profesores y guardar en Neodatis
      InsertarProfesores(conexion);
      // Llenar el set de profesores de asignaturas, por cada objeto
      // asignatura hacemos la select
      llenarSetProfesAsignaturas(conexion);
      // Llenar el set de profesores de Centros y el director
      llenarSetProfesEnCentrosYDirector(conexion);
      conexion.close();
      bd.close();
      } catch (ClassNotFoundException cn) {
                cn.printStackTrace();
      } catch (SQLException e) {
                e.printStackTrace();
      }
}
```

Los métodos son los siguientes:

- **Métodos para comprobar que los objetos existen ya en la BD Neodatis**, se trata de no duplicar objetos si ejecutamos varias veces el programa. Los métodos recibirán el código de asignatura, de profesor y de centro, y devolverán true si existe y false si no existe, los métodos son estos:

```
private static boolean comprobarasig(String cod) {
  try {
        IQuery consulta = new CriteriaQuery(C1Asignaturas.class,
                          Where.equal("codAsig", cod));
        C1Asignaturas obj = (C1Asignaturas)
               bd.getObjects(consulta).getFirst();
        return true;
      } catch (IndexOutOfBoundsException e) {
        return false;}
}

private static boolean comprobarcentro(int cod) {
  try {
        IQuery consulta = new CriteriaQuery(C1Centros.class,
             Where.equal("codCentro", cod));
        C1Centros obj = (C1Centros) bd.getObjects(consulta).getFirst();
        return true;
      } catch (IndexOutOfBoundsException e) {
          return false;}
}

private static boolean comprobarprofe(int cod) {
try {
      IQuery consulta = new CriteriaQuery(C1Profesores.class,
              Where.equal("codProf", cod));
      C1Profesores obj = (C1Profesores)
             bd.getObjects(consulta).getFirst();
      return true;
      } catch (IndexOutOfBoundsException e) {
                return false;}
}
```

- **Métodos para insertar los objetos**, se van a crear tres métodos, uno para insertar las asignaturas, otro para insertar los centros y otro para insertar los profesores. Antes de insertar en la BD Neodatis se comprobará si los objetos existen en la base de datos. Al insertar las asignaturas y los centros el set de profesores se creará vacío, y una vez que se inserten los profesores, se llenarán éstos *set*. Al insertar los profesores sí que se cargará el objeto centro, ya que los centros se habrán insertado en la BD Neodatis. Los métodos son los siguientes:

```
private static void InsertarAsignaturas(Connection conexion) {
try {
   Statement sentencia = (Statement) conexion.createStatement();
   ResultSet resul = sentencia.executeQuery
           ("SELECT * FROM c1_asignaturas");
   while (resul.next()) {
    if (comprobarasig(resul.getString(1)) == false) {
     HashSet<C1Profesores> setprofesores = new HashSet<C1Profesores>();
     C1Asignaturas ass = new C1Asignaturas(resul.getString(1),
```

```
            resul.getString(2), setprofesores);
        bd.store(ass);
        System.out.println("Asignatura grabada " + resul.getString(1));
        } else
         System.out.println("Asig: "+ resul.getString(1)+ ", EXISTE.");
      }
         bd.commit();
         resul.close();sentencia.close();
} catch (SQLException e) {e.printStackTrace();}
}

private static void InsertarCentros(Connection conexion {
try {
    Statement sentencia = (Statement) conexion.createStatement();
    ResultSet resul = sentencia.executeQuery
        ("SELECT * FROM c1_centros");
    while (resul.next()) {
      if (comprobarcentro(resul.getInt(1)) == false) {
         HashSet<C1Profesores> setprofesores = new
             HashSet<C1Profesores>();
         C1Centros cen = new C1Centros(resul.getInt(1),
            resul.getString(2), null, resul.getString(4),
            resul.getString(5), resul.getString(6), setprofesores);
         bd.store(cen);
         System.out.println("Centro grabado " + resul.getInt(1));
       } else
         System.out.println("Centro: " +resul.getInt(1)+ ", EXISTE.");
      }
      bd.commit();
      resul.close();sentencia.close();
} catch (SQLException e) {e.printStackTrace();}
}

private static void InsertarProfesores(Connection conexion) {
try {
      Statement sentencia = conexion.createStatement();
      ResultSet resul = sentencia.executeQuery
           ("SELECT * FROM c1_Profesores");
      while (resul.next()) {
        if (comprobarprofe(resul.getInt(1)) == false)
        {  IQuery consulta = new CriteriaQuery(C1Centros.class,
              Where.equal("codCentro", resul.getInt(6)));
           //Cargamos el centro del profesor
           C1Centros cen = (C1Centros)
             bd.getObjects(consulta).getFirst();
           C1Profesores nueprof = new C1Profesores(
              resul.getInt(1), resul.getString(2), resul.getString(3),
              resul.getDate(4), resul.getString(5), cen);
           bd.store(nueprof);
           System.out.println("Profe grabado " + resul.getInt(1));
         } else
           System.out.println("Profe: "+resul.getInt(1)+", EXISTE.");
      }
      bd.commit();
      resul.close();sentencia.close();
```

```
} catch (SQLException e) {e.printStackTrace();  }
}
```

- **Finalmente se crean los métodos para cargar los set de profesores**. Lo que se hace es recorrer todos los objetos de la clase correspondiente (*C1Asignaturas* o *C1Centros*) y se hace SELECT a la BD relacional, seleccionando los profesores que impartan la asignatura, o que pertenezcan al centro. Y luego se busca el objeto profesor en la BD Neodatis que se corresponda con los códigos de profesor devueltos por la SELECT, y se añade al set de registros. Los métodos son estos:

```
private static void llenarSetProfesAsignaturas(Connection conexion)
throws SQLException {
  Objects<C1Asignaturas> objects = bd.getObjects(C1Asignaturas.class);
  while (objects.hasNext()) {
     C1Asignaturas asi = objects.next();
     HashSet<C1Profesores> setprofesores=new HashSet<C1Profesores>();
     Statement sentencia = conexion.createStatement();
     ResultSet resul = sentencia.executeQuery
         ("SELECT * FROM c1_asigprof where cod_asig = '" +
                                    asi.getCodAsig() + "'");
     while (resul.next()) {
       IQuery consulta = new CriteriaQuery(C1Profesores.class,
            Where.equal("codProf", resul.getInt(2)));
       //Cargo el objeto profesor
       C1Profesores obj = (C1Profesores)
           bd.getObjects(consulta).getFirst();
       //Lo añado al set de profesores
       setprofesores.add(obj);
     }
     //Asigno el set a la asignatura
     asi.setSetprofesores(setprofesores);
     bd.store(asi);
     resul.close();sentencia.close();
  }
  bd.commit();
}

private static void llenarSetProfesEnCentrosYDirector(Connection
conexion) throws SQLException {
      Objects<C1Centros> objectscen = bd.getObjects(C1Centros.class);
      while (objectscen.hasNext()) {
        C1Centros cee = objectscen.next();
        HashSet<C1Profesores> setprofesores = new
             HashSet<C1Profesores>();
        Statement sentencia = conexion.createStatement();
        ResultSet resul = sentencia.executeQuery
           ("SELECT * FROM c1_profesores where cod_centro=" +
             cee.getCodCentro());
        while (resul.next()) {
            IQuery consulta = new CriteriaQuery(C1Profesores.class,
                 Where.equal("codProf", resul.getInt(1)));
            C1Profesores obj = (C1Profesores)
                      bd.getObjects(consulta).getFirst();
            setprofesores.add(obj);
         }
```

```
            //Asigno el set al centro
            cee.setSetprofesores(setprofesores);
            // Localizo al director.
            sentencia = conexion.createStatement();
            resul = sentencia.executeQuery
               ("SELECT director FROM c1_centros where cod_centro=" +
                     cee.getCodCentro());
          if (resul.next()) {
               IQuery consulta = new CriteriaQuery(C1Profesores.class,
                    Where.equal("codProf", resul.getInt(1)));
           try {
                 C1Profesores obj = (C1Profesores)
                       bd.getObjects(consulta).getFirst();
                 cee.setDirector(obj);
               } catch (IndexOutOfBoundsException ee) {
                    System.out.println("Centro  " + cee.getCodCentro() +
                        ", Sin Director, es null.");}
          }
          bd.store(cee);
          resul.close();sentencia.close();
      }
      bd.commit();
}
```

ACTIVIDAD 4.14

Dada la BD Neodatis con nombre ARTICVENTAS.DAT (se adjunta código para crear la BD, en los recursos del tema), cuya estructura de clases es la siguiente:

```
public class Articulos {
      private int codarti;
      private String denom;
      private int stock;
      private float pvp;
      private Set<Ventas> Compras;
.............. }
```

```
public class Clientes {
private int numcli ;
private String nombre;
private String pobla;
.............. }

public class Ventas {
      private int codventa;
      private Clientes numcli ;
      private int univen;
      private String fecha;
.............. }
```

Donde: la clase *Artículos* contiene un set con las *Ventas* de ese artículo, y el cliente que compró el artículo. Se pide leer los datos de esa base de datos y obtener un listado con las ventas de cada artículo. Para cada artículo hay que calcular:

SUMA DE UNIDADES VENDIDAS (SUMA_UNIVEN) es la suma de las unidades vendidas del artículo, las unidades vendidas se encuentran en el set de Ventas de cada artículo.

SUMA DE IMPORTE (SUMA_IMPORTE), es el resultado de multiplicar la suma de las unidades vendidas * el PVP del artículo.

NÚMERO DE VENTAS (NUM_VENTAS), es el contador de ventas del artículo, es decir el número de elementos que aparecen en el Set.

También se desea obtener una línea de totales con la suma de todas las columnas numéricas. La salida del informe debe ser similar a la que se muestra en la Figura 4.5.