1 INTRODUCCIÓN

Los fabricantes de SGBD relaciónales 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-relaciónales.

2 BASES DE DATOS OBJETO-RELACIONALES

Las Bases de Datos Objeto-Relaciónales (BDOR) son una extensión de las bases de datos relaciónales 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.

2.1 CARACTERÍSTICAS

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.
- 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 relaciónales tradicionales.
- 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.

2.2 TIPOS DE OBJETOS

Para crear tipos de objetos utilizamos la orden **CREATE TYPE**. 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. 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);
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';
END;
 /
```

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

Actividad 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.

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.

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.
- **CONSTRUCTOR:** sirve para inicializar el objeto.

Por cada objeto existe un constructor predefinido por Oracle, no obstante podemos sobreescribirlo y/o crear otros constructores adicionales. Los constructores llevarán en la cláusula RETURN la expresión *RETURN SELF AS RESULT.*

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 y un constructor que recibe 2 parámetros:

```
CREATE OR REPLACE TYPE RECTANGULO AS OBJECT
(
   BASE NUMBER,
   ALTURA NUMBER,
   AREA NUMBER,
   CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
      RETURN SELF AS RESULT.
    );
    /
```

Una vez creado el tipo con la especificación de los métodos crearemos el cuerpo del nuevo tipo OBJECT mediante 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;

```
 [STATIC | MEMBER] PROCEDURE nombreProc [ (parametro1, parámetro2, ...) ]
IS
      Declaraciones;
BEGIN
      Instrucciones;
END;
```

```
 [STATIC  |  MEMBER  |  CONSTRUCTOR) FUNCTION nombreFunc
                    [ (parametro1, parámetro2, ... ) ] 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 inicializar el objeto:

```
DECLARE
   DIR DIRECCION := DIRECCION(NULL,NULL,NULL);
    BEGIN
   DIR.SET_CALLE('La Mina, 3') ;
   DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE) ;
    END;
    /
```

La implementación del método constructor del objeto RECTANGULO es la siguiente:

```
CREATE OR REPLACE TYPE BODY RECTANGULO AS
   CONSTRUCTOR FUNCTION RECTANGULO   (BASE NUMBER, ALTURA NUMBER)
      RETURN SELF AS RESULT
   AS
   BEGIN
      SELF.BASE := BASE;
      SELF.ALTURA := ALTURA;
      SELF.AREA := BASE * ALTURA;
      RETURN;
        END;
    END;
    /
```

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 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;
END;
/
```

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 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.**

- Los métodos **MAP** consisten en una función que devuelve un valor de tipo escalar (CHAR, VARCHAR2, NUMBER, DATE,...) 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. No lo trataremos en este tema.

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.**

2.3. TABLAS DE 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 espártales 25',   'GUADALAJARA', 19004),
    '18/12/1987'
);
```

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. 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 N C1 LOOP
      DBMS_OUTPUT.PUT_LINE(I.NOMBRE ||     ' - Calle: ' || I.DIREC.CALLE);
   END LOOP;
    END;
    /
```

Actividad 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.

2.4 TIPOS COLECCIÓN

Las bases de datos relaciónales 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 permiten almacenar un conjunto de elementos, y de las tablas anidadas que permiten almacenar en una columna de una tabla otra tabla.

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);
```

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:='*';
         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: 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.

2.4.2 TABLAS ANIDADAS

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.** El siguiente ejemplo crea un tipo tabla anidada que almacenará objetos del tipo DIRECCION:

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

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)
        )
    );
```

Seleccionamos todas las filas: SELECT * FROM EJEMPLO_TABLA_ANIDADA;

Obtenemos el identificador, los apellidos y las calles de cada fila de la tabla, se obtienen tantas filas como filas hay en la tabla:

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

Se pueden usar cursores dentro de una SELECT para acceder o poner condiciones a las filas de una tabla anidada.

Obtenemos 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 T.DIREC FROM EJEMPLO_TABLA_ANIDADA T WHERE ID=1)
   WHERE CIUDAD='GUADALAJARA';
```

La cláusula **THE** también sirve para seleccionar filas en una tabla anidada, la sintaxis es: *SELECT ... FROM THE (subconsulta) WHERE...*

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', 19005));
```

El operador **TABLE** se utiliza para acceder a la fila que nos interesa, en este caso la que tiene ID=1.

Modificamos la primera dirección del identificador 1:

```
UPDATE TABLE (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET VALUE(PRIMERA) = DIRECCION ('C/Pilon 11',     'TOLEDO', 45589)
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los manantiales 5',   'GUADALAJARA', 19005);
```

El alias PRIMERA recoge los datos devueltos por la SELECT, que debe devolver una fila. Para obtener el objeto almacenado en una fila se necesita la función **VALUE**.

Eliminamos la segunda dirección del identificador 1:

```
DELETE FROM TABLE (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los.manantiales 10',   'GUADALAJARA', 19005);
```

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;
/
BEGIN
   VER_DIREC(1);
END;
/
```

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

2.5 REFERENCIAS

Mediante el operador **REF** asociado a un atributo se pueden definir referencias a otros 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.

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';
```

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:

```
 SELECT NOMBRE, DEREF(P.JEFE) FROM EMPLEADO P;
```

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

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

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,
  MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2,
  MEMBER PROCEDURE VER_DATOS
) NOT FINAL;
/

– 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(
  CURSO VARCHAR2(10),
  NOTA_FINAL NUMBER,
  MEMBER FUNCTION NOTA RETURN NUMBER,
  OVERRIDING MEMBER PROCEDURE VER_DATOS
);
/
— 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
  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
  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:

```
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;
```

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 se muestra el modelo de datos para las tablas CLIENTES, PRODUCTOS, VENTAS y LINEASVENTAS. Las órdenes de creación de las tablas son:

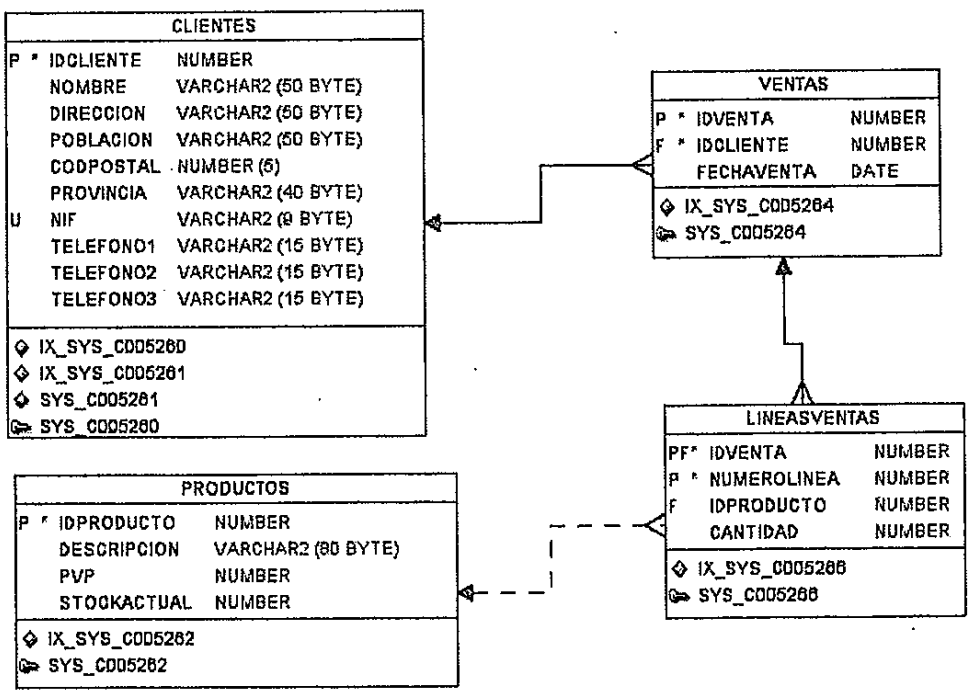
```
CREATE TABLE CLIENTES (
  IDCLIENTE       NUMBER PRIMARY KEY
  NOMBRE          VARCHAR2(50),
  DIRECCION       VARCHAR2(50),
  POBLACION       VARCHAR2(50),
  CODPOSTAL       NUMBER(5),
  PROVINCIA       VARCHAR2(40) ,
  NIF             VARCHAR2(9) UNIQUE,
  TELEFONO1       VARCHAR2(15),
  TELEFONO2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

);
```



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_PRODUCTO AS OBJECT (
IDPRODUCTO      NUMBER,
DESCRIPCION     VARCHAR2(80),
PVP             NUMBER,
STOCKACTUAL     NUMBER
);
/

CREATE TYPE TIP_CLIENTE AS OBJECT(
IDCLIENTE       NUMBER,
NOMBRE          VARCHAR2(50),
DIREC           TIP_DIRECCION,
NIF             VARCHAR2(9),
TELEF           TIP_TELEFONOS
);
/

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);
```

Inserto 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;
```

Inserto 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);
```

Inserto 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;
```

Inserto 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);
```

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

```
CREATE OR REPLACE PROCEDURE VER_VENTA (ID NUMBER) AS
   LIN NUMBER;
   CANT NUMBER;
   IMPORTE NUMBER;
   TOTAL_V NUMBER;
   PRODUC TIP_PRODUCTO:=TIP_PRODUCTO(NULL,NULL,NULL,NULL);
   CLI TIP_CLIENTE:=TIP_CLIENTE(NULL,NULL,NULL,NULL, NULL);
   DIR TIP_DIRECCION:=TIP_DIRECCION(NULL,NULL,NULL,NULL);
   FEC DATE;
   CURSOR C1 IS
      SELECT NUMEROLINEA, DEREF(IDPRODUCTO), CANTIDAD FROM THE
            (SELECT T.LINEAS FROM TABLA_VENTAS T WHERE IDVENTA=ID);
BEGIN
   SELECT DEREF (IDCLIENTE), FECHAVENTA, V.TOTAL_VENTA()
   INTO CLI, FEC, TOTAL_V
   FROM TABLA_VENTAS V WHERE IDVENTA=ID;
   DIR :=CLI.DIREC;
   DBMS_OUTPUT.PUT_LINE('NUMERO DE VENTA:    '||ID||'    ' * Fecha de venta:    '|| FEC);
   DBMS_OUTPUT.PUT_LINE('CLIENTE:   '||CLI.NOMBRE);
   DBMS_OUTPUT.PUT_LINE( 'DIRECCION:   '| |DIR.CALLE);
   DBMS_OUTPUT.PUT_LINE('==========================================') ;

   OPEN C1;
   FETCH C1 INTO LIN, PRODUC, CANT;
   WHILE C1%FOUND LOOP
      IMPORTE:= CANT * PRODUC.PVP;
      DBMS_OUTPUT.PUT_LINE (LIN||'*'||PRODUC.DESCRIPCION ||'*'||PRODUC.PVP||'*'||CANT||'*'||IMPORTE);
      FETCH C1 INTO LIN, PRODUC, CANT;
   END LOOP;
   CLOSE C1;
   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;
/
```

```
NUMERO DE VENTA: 2 * Fecha de venta: 19/03/12
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 5: Crea una función almacenada que reciba un identificador de venta y retorne el total de la venta. Realiza un bloque PL/SQL anónimo que haga uso de la función.

3. BASES DE DATOS ORIENTADAS A OBJETOS

Las Bases de Datos Orientadas a Objetos (BDOO) son aquellas cuyo modelo de datos está orientado a objetos. 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.

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 usuados 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): **encapsulacion, identidad, herencia y polimorfismo**.

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 encapsulacion: 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.

3.2 EL ESTÁNDAR ODMG

ODMG *(Object Datábase 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 especifíca 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-Object IDentifier)* 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 equivalentes a una clase concreta en los lenguajes de programación. Una clase es un tipo de objetos asociado a un *"extent".*

El lenguaje **ODL** es el equivalente al lenguaje de definición de datos (DDL) de los SGBD tradicionales. 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 Dirección{
                                    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 pertence 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();
```

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 tipicos>
    [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, short 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.descripción, 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.descripción,
        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, dpw.string, can: short, imp: float)).*

3.3.1 OPERADORES DE COMPARACIÓN

Lo valores pueden ser comparados usando los siguientes operadores:

> 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 cadena.2. cadena.2* puede contener caracteres especiales:
  _ o ?: indicador de posición, representa cualquier carácter.
  * o %: representa una cadena de caracteres.

Ejemplo: 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.descripción, lin.cantidad , lin.importe()
FROM Ventas v, v.lineas lin
WHERE v.IDVENTA=1 AND 'P' IN lin.product.descripción;
```

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 Pen-drive 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 los 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)).*

4 EJEMPLO DE BDOO

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-1.930.689.zip.* Lo descomprimimos y copiamos el fichero *neodatis-odb-1.930.689.jar* en la carpeta adecuada para después definirlo en el CLASSPATH o incluirlo en nuestro proyecto Eclipse.

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;
	prívate 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 getNombre0 {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.println(objects.size() + " Jugadores:");

           int i = 1;
           // visualizar los objetos
           while(objects.hasNext() ){
                Jugadores jug = objects.next();
                System.out.println((i++) + "\t: " + jug.getNombre() + "*" +jug.getDeporte()+"*" + jug.getCiudad()+"*"+ jug.getEdad());
             }
           odb.close(); // Cerrar BD
      }
}
```

Actividad 6: 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 Países 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;}
```

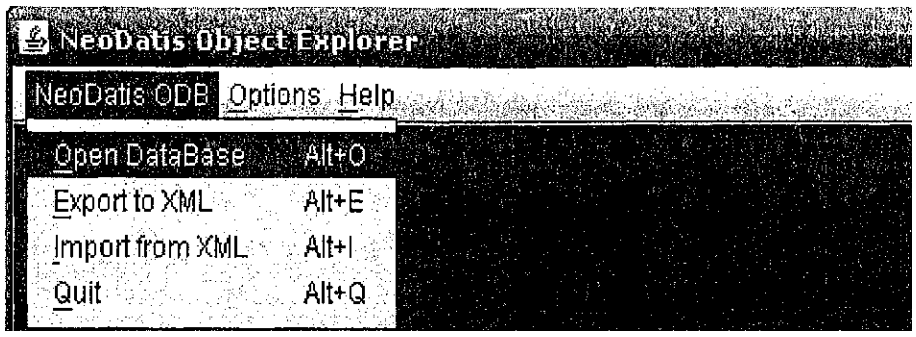
Crea la clase Jugadores (en el paquete *clases,* como en el ejemplo anterior) y añade el siguiente atributo con sus *getter y setter.*

```
private Paises pais;
```

Crea una clase Java (con el método main) que cree una base de datos de nombre EQUIPOS. DB e inserta países y los jugadores de esos países.

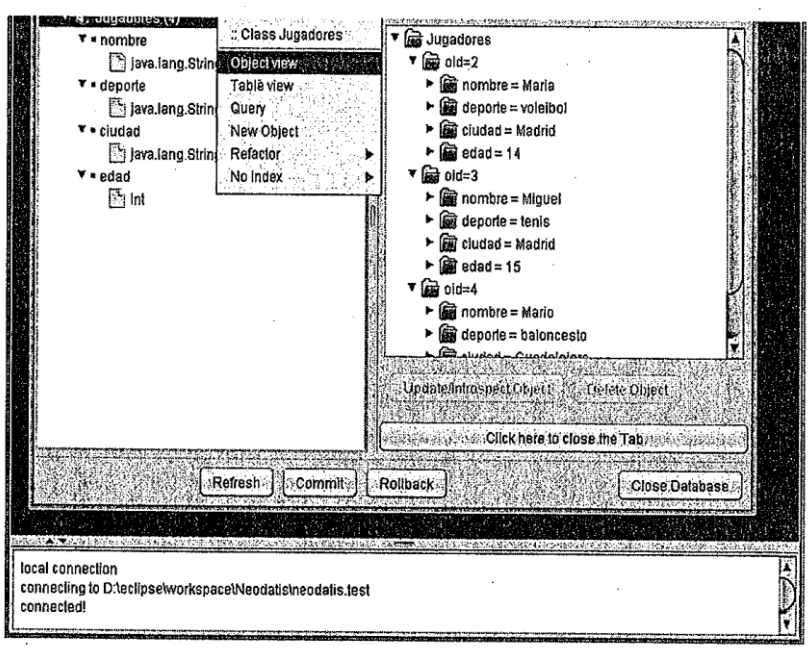
Añade otra clase Java para visualizar los países y los jugadores que hay en la BD.

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). Es necesario abrir la base de datos por la que vamos a navegar, pulsamos en el menú *NeoDatis ODB-> Open Datábase*



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.

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 Datábase.*



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.println( jug.getNombre() + "*" +     jug.getDeporte()+ "*" + jug.getCiudad()+ "*" + jug.getEdad() );
         odb.close();                                                 // Cerrar BD
   }
}
```

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.* Ejemplo, para obtener el **OID** del objeto j1 podemos hacerlo de las siguientes maneras:

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

4.1 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():*

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

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 7: 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().* No olvidemos que para validar los cambios es necesario cerrar la BD. El siguiente ejemplo cambia el deporte de Maria a vóley-playa:

```
query = new CriteriaQuery (Jugadores.class, Where.egual("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.close();                                   // 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);
```

4.2 CONSULTAS MÁS COMPLEJAS

Para utilizar agrupamientos GROUP BY y las funciones de grupo SUM, MAX, MIN, AVG y COUNT usamos la API *Object Values* de Neodatis que provee acceso a los atributos de los objetos. También se puede usar para recuperar objetos.

Por ejemplo, para obtener el nombre y la ciudad de todos los jugadores usando esta API escribimos el siguiente código:

```
ODB odb = ODBFactory.open("neodatis.test").;                    // Abrir BD
Values valores = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).field("nombre") .field("ciudad")) ;
while(valores.hasNext()){
        ObjectValues objectValues= (ObjectValues) valores.next();
        System.out.println(ObjectValues.getByAlias("nombre")
                + " ciudad " + ObjectValues.getBylndex(1)) ;
}
```

Mediante los métodos *getByAlias("alias")* o *getBylndex(índice)* podemos obtener los valores de los atributos de un objeto. Se puede definir un alias a cada atributo de la clase. En el ejemplo anterior en el método field() no se especificó alias, entonces para recuperar el valor del atributo se indica su nombre en el método *getByAlias().* Si definimos el atributo *nombre* con alias: *field("nombre", "n");* entonces a la hora de recuperar el valor del atributo usamos *getByAlias("n").*

Para trabajar con la API *Object Values* será necesario importar algunos paquetes:

```
import java.math.BigDecimal;
import java.math.Biglnteger;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.Values;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;
```

Veamos algunos ejemplos de uso de las funciones de grupo, se pone comentada la sentencia SQL a la que equivaldrían las líneas de código expuestas:

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

//SUMA - Obtiene la suma de las edades
//SELECT SUM(edad) FROM Jugadores
Values     val = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).sum("edad"));
ObjectValues ov = val.nextValues();
BigDecimal value = (BigDecimal) ov.getByAlias("edad");
System.out.println("Suma de edad : "+ value.longValue());

//CUENTA - Obtiene el número de jugadores
//SELECT COUNT(nombre) FROM Jugadores
Values     val2 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).count("nombre"));
ObjectValues ov2 = val2.nextValues();
Biglnteger value2 = (Biglnteger) ov2.getByAlias("nombre");
System.out.println("Numero de jugadores : "+ value2.intValue());


//MEDIA - Obtiene la edad media de los jugadores
//SELECT AVG(edad) FROM Jugadores
Values     val3 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).avg("edad"));
ObjectValues ov3 = val3.nextValues ();
BigDecimal value3 = (BigDecimal) ov3.getByAlias("edad");
System.out.println("Edad media : "+ value3.floatValue());


//MAXIMO Y MINIMO - Obtiene la edad máxima y la edad minima
//SELECT MAX(edad) edad_max , MIN(edad) edad_min FROM Jugadores
Values   val4 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).max("edad","edad_max").min("edad","edad_min"));
ObjectValues ov4 = val4.nextValues();
BigDecimal maxima = (BigDecimal) ov4.getByAlias("edad_max");
BigDecimal minima = (BigDecimal) ov4.getByAlias("edad_min");

System.out.println("Edad máxima: "+ maxima.intValue()+"    Edad minima: "+ minima.intValue());
```

En este ejemplo usamos GROUP BY, obtenemos por cada ciudad el número de jugadores:

```
Values     groupby = odb.getValues(new ValuesCriteriaQuery(Jugadores.class) .field("ciudad").count("nombre").groupBy("ciudad"));

while(groupby.hasNext()) {
        ObjectValues objetos= (ObjectValues) groupby.next();
        System.out.println(objetos.getByAlias("ciudad") + " * "+objetos.getByIndex(1));
}
```

En SQL la consulta sería: *SELECT ciudad, count(nombre) FROM Jugadores GROUP BY ciudad.*

Cuando un objeto está relacionado con otro, por ejemplo, un jugador está relacionado con su país, podremos acceder a la información del objeto relacionado para obtener la información necesaria gracias a la API *Object Valúes. A* esta característica se le da el nombre de vistas dinámicas, y son como los joins en SQL.

Por ejemplo, partimos de las clases Países y Jugadores (creadas en la Actividad 6), vamos a obtener el nombre, edad y país del jugador; en SQL sería como realizar la siguiente consulta: *SELECT nombre, edad, nombrepais FROM Jugadores j, Paises p WHERE j.id = p.id;*

Veamos como sería la consulta:

```
 Values     valores = odb.getValues(new ValuesCriteriaQuery(Jugadores.class) .field ("nombre") .field ("edad").field("pais.nombrepais"));

while (valores.hasNext()) {
        ObjectValues ObjectValues = (ObjectValues) valores.next();
        System.out.println("Nombre: "+ ObjectValues.getByAlias("nombre") +
                        " Edad: " + ObjectValues.getByIndex(1) +
                        " Pais: " + ObjectValues.getBylndex(2));
 }
```

Para obtener el nombre del país en el *método field()* escribimos la relación completa, el atributo de la clase Jugadores, un punto y a continuación el atributo de la clase Paises*: field("pais.nombrepais").*

Para obtener el nombre y la edad de los jugadores cuyo nombre de pais es ESPAÑA escribimos:

```
Values valores = odb.getValues(new ValuesCriteriaQuery(Jugadores.class, Where.equal("pais.nombrepais","ESPAÑA"))
                .field ("nombre") .field ("ciudad"));
```

Actividad 8: Realiza una consulta para obtener el número de jugadores de España y edad = 15.

4.3 MODO CUENTE/SERVIDOR DE LA BD

NeoDatis ODB también puede ser utilizada como una base de datos cliente/servidor. Lo primero que hemos de hacer es iniciar el servidor y configurar los siguientes parámetros:

- El puerto donde se va a ejecutar y recibir las conexiones de clientes, hemos de asegurarnos que esté libre. Por ejemplo, para crear el servidor en el puerto 8000 escribimos:

```
ODBServer server = ODBFactory.openServer(8000);
```

- La base de datos que debe ser manejada por el servidor. Esto se hace mediante el método *addBase()* en el que se especifica el nombre y el fichero de base de datos; este nombre será utilizado por los clientes. Para indicar que trabajaremos con el fichero de base de datos *neodatist.test* y que los clientes usarán el nombre *base* para dirigirse a ella escribimos:

```
server.addBase("base", "D:/uni4/server/neodatis.test");
```

- El servidor se inicia como un subproceso en segundo plano usando el método *StartServer (true).* Para iniciar el servidor escribimos:

```
server.StartServer(true);
```

A continuación se debe crear el cliente. Hay dos formas; si el cliente se ejecuta en la misma máquina virtual que el servidor se puede crear una instancia del servidor de esta manera:

```
ODB odb = server.openClient("base");
```

Si el cliente se ejecuta en otra máquina virtual, se debe utilizar *ODBFactory:*

```
ODB odb = ODBFactory.openClient("localhost", 8000, "base");
```

Para usar el modo cliente/servidor es necesario añadir el paquete *org.neodatis.odb.ODBServer.* Hay que tener en cuenta que las clases deben implementar *Serializable.*

El siguiente ejemplo muestra el uso del modo cliente/servidor en la misma máquina virtual. La clase *Jugadores* se ha incluido en un fichero aparte. Cuando se trabaja con *NeoDatis ODB,* es importante llamar al método *close()* para confirmar los cambios. Para estar seguros de hacer esto, es una buena práctica utilizar un bloque **try - finally:**

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.ODBServer;

public class EjemploNeodatisCliente {
   public static void main(String[] args) {
      ODB odb = null;
      ODBServer server = null;
      try {
         // Crea el servidor en el Puerto 8000
         server = ODBFactory.openServer(8000);
         // BD a usar y el nombre que se usará para refererirse a ella
         server.addBase("base", "D:/uni4/server/neodatis.test");
         / / S e  ejecuta el servidor
         server.StartServer(true);

         // Se abre la BD
         odb = server.openClient("base");

         // Se llama al método que visualize los datos
         VisualizaDatos(odb);

      } finally {
            if (odb != null) {
               // Primero se cierra el cliente
               odb.close();
            }
            if (server != null) {
            / / Y  luego el servidor
            server.close();
            }
      }
   }// fin main
/ /
   static void VisualizaDatos(ODB odb){
      //recuperamos todos los objetos
      Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
      System.out.println(objects.size() + " Jugadores:");

      int i = 1;
      // visualizar los objetos
      while(objects.hasNext()){
               Jugadores jug = objects.next();
               System.out.println((i++) + "\t: " + jug.getNombre() + "*" + jug.getDeporte()+ "*" + jug.getCiudad()+ "*" + jug.getEdad());
    }
  }
}// fin clase
```

A continuación se muestra el resultado de la ejecución, antes hemos de añadir el fichero JAR *(neodatis-odb-1.9.30.689.jar)* al CLASSPATH y compilar las clases **Jugadores.java** y **EjemploNeodatisCliente.java:**

```
D:\uni4\server>set CLASSPATH=.;D:\uni4\server\neodatis-odb-1.9.30.689.jar

D:\uni4\server>javac   Jugadores.java

D:\uni4\server>javac    EjemploNeodatisCliente.java

D:\uni4\server>java EjemploNeodatisCliente
NeoDatis ODB Server [version=1.9.30 - build=689-10-ll-2010-08-21-21] running on port 8000
Managed bases: [base] 4 Jugadores:
1           : Maria*voleibol*Madrid*14
2           : Miguel*tenis*Madrid*15
3           : Mario*baloncesto*Guadalajara*15
4           : Alicia*tenis*Madrid*14

NeoDatis ODB Server (port 8000) shutdown [uptime=0Hours] D:\uni4\server>
```

El siguiente ejemplo muestra el modo cliente/servidor ejecutándose el servidor y el cliente en distintas máquinas virtuales. Por un lado tenemos el proceso servidor que se ejecutará en segundo plano:

```
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.ODBServer;

public class EjemploServer{
   public static void main(String[] args) {
      ODBServer server = null;
       server = ODBFactory.openServer(8000);                    //Crea el servidor en el puerto 8000
       server.addBase("base1", "D:/uni4/server/neodatis.test");  // Abrir BD
       server.StartServer(true);                                 / / S e  inicia el servidor ejecutándose en segundo plano
    }
}
```

Lo compilamos y ejecutamos desde la línea de comandos del DOS, permanece ejecutándose hasta que cerremos la ventana del DOS o paremos el proceso (pulsando las teclas [Ctrl+C]):

```
D:\uni4\server>javac EjemploServer.java

D:\uni4\server>java EjemploServer
NeoDatis ODB Server [version=1.9.30 - build=689-10-11-2010-08-21-21]   running on port 8000
Managed bases: [base1]
```

El siguiente código Java muestra el cliente que se ejecutará en otra máquina virtual, se insertará un nuevo jugador en la base de datos. Se debe abrir una nueva consola DOS para ejecutarlo (no olvide añadir el JAR al CLASSPATH):

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

public class EjemploCliente {
   public static void main(String[] args) {
      ODB odb = null;
      try {
         odb = ODBFactory.openClient("localhost",8000,"base1");
         Jugadores j4 = new Jugadores("Andrea","padel","Guadalajara", 14);
         odb.store (j4);
      } finally {
            if (odb != null) {odb.close();}
       }
    )//—main
}
```

Lo compilamos y ejecutamos desde la línea de comandos del DOS:

```
d:\uni4\server>javac   EjemploCliente.java
d:\uni4\server>java EjemploCliente
d:\uni4\server>
```

5 EJERCICIOS

1. Responde a las siguientes cuestiones sobre BD objeto-relacional:

A) La orden CREATE TYPE:

a)Permite definir distintos tipos de tablas.

b)Permite definir tipos de objetos.

c)Permite definir restricciones a los tipos de datos.

d)Ninguna es correcta.

B) Los modelos de datos relaciónales orientados a objetos:

a)Extienden el modelo de datos relacional proporcionando tipos de datos complejos y la programación orientada a objetos.

b)No permiten el uso de SQL como lenguaje de consulta ya que hay tipos complejos de datos.

c)Respuestas a) y b) correctas.

d)Ninguna de las anteriores.

C) ¿Cuál de estas afirmaciones es correcta?

a) Los lenguajes de consulta relaciónales como SQL necesitan ser extendidos para trabajar con tipos de datos orientados a objetos.

b)En las BBDD relaciónales orientadas a objetos se encuentran extensiones orientadas a objetos, como los tipos de datos definidos por el usuario.

c)En las BBDD relaciónales orientadas a objetos no se encuentran extensiones sobre las tablas anidadas.

d)Respuestas a) y b) correctas.

2. Realiza las siguientes operaciones en Oracle:

- Crea un tipo de objeto denominado SITUACION. Este tipo contiene las columnas NUMEROEDIFICIO de tipo NUMBER(4) y CIUDAD de tipo VARCHAR2(30).
- Crea un VARRAY de nombre EMAIL para cinco elementos.
- Crea un tipo de nombre DATOSEMPLE con las siguientes columnas: IDENTIFICADOR de tipo NUMBER(4), NOMBRE de tipo VARCHAR2(30), DIRECCION de tipo VARCHAR2(40), OFICINA de tipo SITUACION, CORREOS de tipo EMAIL. Crea una tabla de nombre UNIDAD4 con dos columnas: CLAVE, de tipo NUMBER(3) clave primaria y DATOS del tipo DATOSEMPLE definido anteriormente.
- ¿Cuál de las siguientes sentencias para insertar una fila es correcta?:

```
a)  INSERT INTO UNIDAD4 VALUES(22,DATOSEMPLE (1,'MANUEL', 'PILON 10',SITUACION (1,'MADRID'), EMAIL('a@uno.es",'b@uno.es ' )));
b)  INSERT INTO UNIDAD4 VALUES(22, DATOS(. 1,'MANUEL', 'PILON 10',SITUACION(1,'MADRID'), CORREOS('a@uno.es','b@uno.es' )));
c)  INSERT INTO UNIDAD4   (CLAVE,DATOS) VALUES(22,DATOSEMPLE (1,'MANUEL', 'PILON10', SITUACION(1,'MADRID'),
EMAIL('a@uno.es', 'b@uno.es' )));
```

3. ¿Cuál de las siguientes afirmaciones sobre ODMG es correcta?

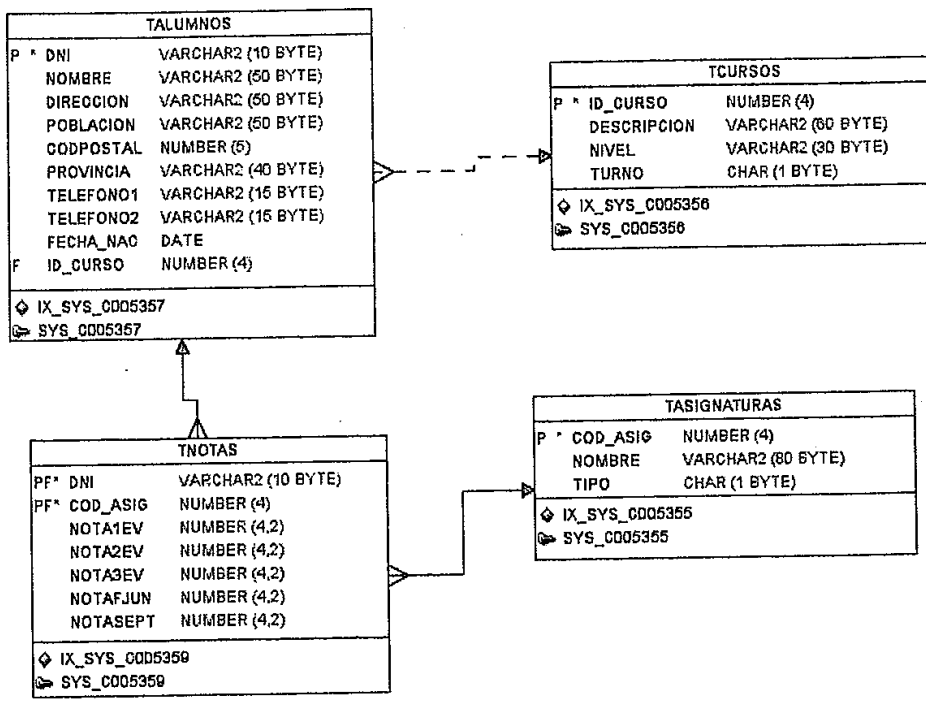
a) ODMG es un grupo formado por fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. La última versión del estándar del mismo nombre propone 3 componentes: el modelo de objetos, el lenguaje ODL y el lenguaje OQL.

b) 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.

c) El lenguaje OQL no incluye operaciones de actualización, solo son de consulta. Las actualizaciones se realizan mediante los métodos que los objetos poseen.

d) OQL no dispone de los operadores sobre colecciones para obtener el valor máximo, el mínimo, la cuenta... en su lugar se utilizan los métodos definidos en el objeto.

4. Construye a partir del modelo relacional de la Figura el modelo de objetos. Se dispone de 4 tablas: TALUMNOS con información de alumnos, TCURSOS con información de cursos, TASIGNATURAS con información de asignaturas y TNOTAS con información de notas en cada asignatura.



En la tabla TNOTAS tenemos las notas que tiene el alumno en cada asignatura. Existen tantas filas como asignaturas tenga el alumno. Las columnas NOTA1EV, NOTA2EV,... corresponden a la nota de la 1ª, 2ª y 3ª evaluación, la final y la nota de septiembre (puede no tener nota en septiembre si lo aprueba todo).

Crea en Oracle los tipos necesarios. Por ejemplo: un tipo array para guardar las 5 notas, otro tipo array para los teléfonos, un tipo para almacenar la dirección, un tipo tabla anidada que contenga la asignatura y la nota del alumno. Define también un tipo alumno, este deberá tener la tabla anidada con las notas, un tipo para cursos y un tipo asignaturas.

En el nuevo modelo de objetos se deberán crear sólo 3 tablas, una para alumnos, otra para asignaturas y otra para cursos.

Realiza después un procedimiento almacenado que reciba un DNI de alumno y visualice el nombre, la dirección y las notas obtenidas en cada asignatura y en cada evaluación. Prueba el procedimiento.

5. Partimos de las clases Departamentos y Empleados, similares a las definidas en unidades anteriores. Los atributos son los siguientes:

Departamentos:	Empleados:
private int deptNo; **private** String dnombre; **private** String loe;	**private int** empNo; **private** String apellido; **private** String oficio; **private** Empleados dir; **private** java.sql.Date fechaAlt; **private float** salario; **private float** comision; **private** Departamentos dept;

Define los métodos *getter* y *setter* para cada atributo y el método *toString()* para que devuelva el nombre del departamento en la clase Departamentos y el nombre del empleado en la clase Empleados. Definir también los constructores necesarios. Todos los empleados deben tener un director. El empleado de mayor rango (el presidente) tiene como director a él mismo.

Crea una base de datos en Neodatis para guardar datos de empleados y departamentos. Hazlo como un proyecto Eclipse, dentro del proyecto crea un paquete para las clases anteriores. Realiza las siguientes clases Java dentro del proyecto:

1. Una clase que inserte objetos en la base de datos.
2. Clase para visualizar todos los departamentos y empleados de la base de datos. Por cada empleado visualizar el nombre de su director y el de su departamento.
3. Clase donde se realizan diferentes consultas:
   a) Apellidos de los empleados del departamento 10.
   b) Número de empleados del departamento de VENTAS.
   c) Apellido de los empleados cuyo director es FERNÁNDEZ.
   d) Por cada departamento el número de empleados.