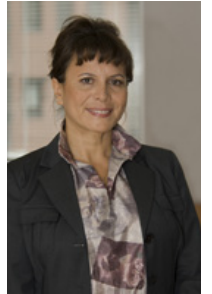


## AD05.- Bases de datos objeto-relacionales y orientadas a objetos.

### Caso práctico

Esta mañana **Juan, María y Ana** han quedado a primera hora en el despacho de **Ada**. Según les comentó ayer **Ada**, antes de cerrar, urge comenzar a planificar un proyecto nuevo que ha llegado a la empresa. Se trata de desarrollar una aplicación informática para una empresa de la industria de la madera. La empresa se dedica a la fabricación de mobiliario de diseño, así como a su distribución y venta.

La aplicación que desarrollen debe gestionar un gran volumen de piezas diferentes, piezas de chapa y madera, permitiendo diseñarlas desde la forma más simple a la más compleja. También, debe permitir desplegar y gestionar directamente la evolución de la pieza añadiendo pliegues, cortes, deformaciones, etc. Por descontado, debe gestionar también el almacenamiento de todas esas piezas y los productos finales.



**Ada y Juan** tienen claro que en esta aplicación no tiene cabida el uso de bases de datos relacionales, ya que habrá que almacenar un gran volumen de datos diferentes, con propiedades y comportamientos que no son fijos, y con complejas relaciones entre ellos. Será necesario un tipo de bases de datos más avanzado, posiblemente orientadas a objetos u objeto-relacionales, sin las limitaciones que presentan las bases de datos relacionales, entre ellas, que solo son capaces de manejar estructuras muy simples (filas y columnas).

**Ana** ha estado algo callada durante la reunión, sabe que ella va a echar una mano en este proyecto junto a **Antonio**, y necesita repasar muchísimo sobre bases de datos avanzadas. Lo primero que ha decidido hacer es coger sus apuntes del ciclo de DAM e ir echando un vistazo a las características de las bases de datos orientadas a objetos y objeto-relacionales.

## 1.- Introducción.

Las Bases de Datos Relacionales (BDR) son ideales para aplicaciones tradicionales que soportan tareas administrativas, y que trabajan con datos de estructuras simples y poco cambiantes, incluso cuando la aplicación pueda estar desarrollada en un lenguaje OO y sea necesario un Mapeo Objeto Relacional (ORM)

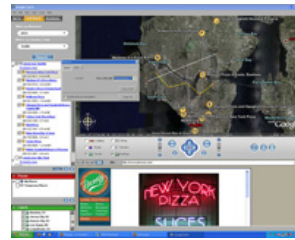
Pero cuando la aplicación requiere otras necesidades, como por ejemplo, soporte multimedia, almacenar objetos muy cambiantes y complejos en estructura y relaciones, este tipo de base de datos no son las más adecuadas. Recuerda, que si queremos representar un objeto y sus relaciones en una BDR esto implica que:

- ✓ Los objetos deben ser descompuestos en diferentes tablas.
- ✓ A mayor complejidad, mayor número de tablas, de manera que se requieren muchos enlaces (joins) para recuperar un objeto, lo cual disminuye dramáticamente el rendimiento.

Las **Bases de Datos Orientadas a Objetos** (BDOO) o Bases de Objetos se integran directamente y sin problemas con las aplicaciones desarrolladas en lenguajes orientados a objetos, ya que **soportan un modelo de objetos puro** y son ideales para almacenar y recuperar datos complejos permitiendo a los usuarios su **navegación** directa (sin un mapeo entre distintas representaciones).

Las Bases de Objetos aparecieron a finales de los años 80 motivadas fundamentalmente por dos razones:

- ✓ Las necesidades de los lenguajes de Bases de Datos Orientadas a Objetos (BDOO), como la necesidad de persistir objetos.
- ✓ Las limitaciones de las bases de datos relacionales, como el hecho de que sólo manejan estructuras muy simples (tablas) y tienen poca riqueza semántica



[.\(link: AD05\\_CONT\\_R03\\_googleEarth.jpg.\)](#)

Pero como las BDOO no terminaban de asentarse, debido fundamentalmente a la inexistencia de un estándar, y las BDR gozaban y gozan en la actualidad de una gran aceptación, experiencia y difusión, debido fundamentalmente a su gran robustez y al lenguaje SQL, los fabricantes de bases de datos comenzaron a implementar nuevas funcionalidades orientadas a objetos en las BDR existentes. Así surgieron, las bases de datos objeto-relacionales.

Las **Bases de Datos Objeto-Relacionales** (BDOR) son bases de datos relacionales que han evolucionado hacia una base de datos más extensa y compleja, incorporando conceptos del modelo orientado a objetos. Pero en estas bases de datos **aún existe un mapeo de objetos subyacente**, que es costoso y poco flexible, cuando los objetos y sus interacciones son complejos.



## 2.- Características de las bases de datos orientadas a objetos.

### Caso práctico

**Ana** se ha llevado a la empresa sus apuntes sobre bases de datos y acceso a datos del ciclo formativo, y ha empezado a repasar con **Juan** las características de las Bases de Objetos.

—¡Pero claro!, —le dice Juan a Ana— habrá que detenerse en valorar tanto las posibles ventajas como los inconvenientes de estas bases de datos, ya que tenemos que elegir el sistema de almacenamiento óptimo para esta aplicación. Ya sabes que no podemos fallar. ¡Somos los mejores! —y le sonríe.



En una BDOO, los datos se almacenan como objetos. Un **objeto** es, al igual que en POO, una entidad que se puede identificar unívocamente y que describe tanto el **estado** como el **comportamiento** de una entidad del 'mundo real'. El estado de un objeto se describe mediante atributos y su comportamiento es definido mediante procedimientos o métodos.

Entonces, ¿a qué equivalen las entidades, ocurrencias de entidades y relaciones del modelo relacional? Las entidades son las clases, las ocurrencias de entidad son objetos creados desde las clases, las relaciones se mantienen por medio de inclusión lógica, y no existen claves primarias, los objetos tienen un identificador.

La principal característica de las BDOO es que **soportan un modelo de objetos puro y que el lenguaje de programación y el esquema de la base de datos utilizan las mismas definiciones de tipos**.



**Otras características importantes** de las BDOO son las siguientes:

- ✓ **Soportan las características propias de la Orientación a Objetos** como agregación, encapsulamiento, polimorfismo y herencia. La herencia se mantiene en la propia base de datos.
- ✓ **Identificador de objeto (OID)**. Cada objeto tiene un identificador, generado por el sistema, que es único para cada objeto, lo que supone que cada vez que se necesite modificar un objeto, habrá que recuperarlo de la base de datos, hacer los cambios y almacenarlo nuevamente. Los OID son independientes del contenido del objeto, esto es, si cambia su información, el objeto sigue teniendo el mismo OID. Dos objetos serán equivalentes si tienen la misma información pero diferentes OID.
- ✓ **Jerarquía y extensión de tipos**. Se pueden definir nuevos tipos basándose en otros tipos predefinidos, cargándolos en una jerarquía de tipos (o jerarquía de clases).
- ✓ **Objetos complejos**. Los objetos pueden tener una estructura de objeto de complejidad arbitraria, a fin de contener toda la información necesaria que describe el objeto.
- ✓ **Acceso navegacional de datos**. Cuando los datos se almacenan en una estructura de red densa y probablemente con una estructura de diferentes niveles de profundidad, el acceso a datos se hace principalmente navegando la estructura de objetos y se expresa de forma natural utilizando las construcciones nativas del lenguaje, sin necesidad de uniones o joins típicas en las BDR.
- ✓ **Gestión de versiones**. El mismo objeto puede estar representado por múltiples versiones. Muchas aplicaciones de bases de datos que usan orientación a objetos requieren la existencia de varias versiones del mismo objeto, ya que si estando la aplicación en funcionamiento es necesario modificar alguno de sus módulos, el diseñador deberá crear una nueva versión de cada uno de ellos para efectuar cambios.

### Autoevaluación

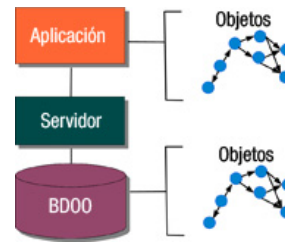
Señala las opciones correctas. Las bases de datos orientadas a objetos:

- ☐ *(link: )*  
Soportan conceptos de orientación a objetos como la herencia.
- ☐ *(link: )*  
Permiten la manipulación navegacional.
- ☐ *(link: )*  
Tienen el mismo tipo de problemas que las relacionales para gestionar objetos complejos.
- ☐ *(link: )*  
Cada objeto posee un identificador de objeto.

## 2.1.- Ventajas e inconvenientes.

El uso de una BDOO puede ser ventajoso frente a una BDR relacional si nuestra aplicación requiere alguno de estos elementos :

- ✓ Un gran número de tipos de datos diferentes.
- ✓ Un gran número de relaciones entre los objetos.
- ✓ Objetos con comportamientos complejos.



[\(link: AD05\\_CONT\\_R05\\_ventajaBDOO.jpg.\)](#)

Una de las principales ventajas de los sistemas de bases de datos orientados a objetos es la **transparencia**, (manipulación directa de datos utilizando un entorno de programación basado en objetos), por lo que el programador, solo se debe preocupar de los objetos de su aplicación, en lugar de cómo los debe almacenar y recuperar de un medio físico.

**Otras ventajas** de un sistema de bases de datos orientado a objetos son las siguientes:

- ✓ **Gran capacidad de modelado.** El modelado de datos orientado a objetos permite modelar el 'mundo real' de una manera óptima gracias al encapsulamiento y la herencia.
- ✓ **Flexibilidad.** Permiten una estructura cambiante con solo añadir subclases.
- ✓ **Soporte para el manejo de objetos complejos.** Manipula de forma rápida y ágil objetos complejos, ya que la estructura de la base de datos está dada por referencias (apuntadores lógicos) entre objetos.
- ✓ **Alta velocidad de procesamiento.** Como el resultado de las consultas son objetos, no hay que reensamblar los objetos cada vez que se accede a la base de objetos
- ✓ **Extensibilidad.** Se pueden construir nuevos tipos de datos a partir de los ya existentes, agrupar propiedades comunes de diversas clases e incluirlas en una superclase, lo que reduce la redundancia.
- ✓ **Mejora los costes de desarrollo,** ya que es posible la reutilización de código, una de las características de los lenguajes de programación orientados a objetos.
- ✓ **Facilitar el control de acceso y concurrencia,** puesto que se puede bloquear a ciertos objetos, incluso en una jerarquía completa de objetos.
- ✓ Funcionan de forma **eficiente en entornos cliente/servidor y arquitecturas distribuidas.**

Pero aunque los sistemas de bases de datos orientados a objetos pueden proporcionar soluciones apropiadas para muchos tipos de aplicaciones avanzadas de bases de datos, también tienen sus **desventajas**. Éstas son las siguientes:

- ✓ **Carencia de un modelo de datos universal.** No hay ningún modelo de datos aceptado universalmente, y la mayor parte de los modelos carecen de una base teórica.
- ✓ **Falta de estándares.** Existe una carencia de estándares general para los sistemas de BDOO.
- ✓ **Complejidad.** La estructura de una BDOO es más compleja y difícil de entender que la de una BDR.
- ✓ **Competencia de otros modelos.** Las bases de datos relacionales y objeto-relacionales están muy asentadas y extendidas, siendo un duro competidor.
- ✓ **Difícil optimización de consultas.** La optimización de consultas requiere una comprensión de la implementación de los objetos, para poder acceder a la base de datos de manera eficiente. Sin embargo, esto compromete el concepto de encapsulación.

## Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

En una BDOO el resultado de una consulta son objetos, por lo que no es necesario reensamblar los objetos cada vez que se accede a la base de datos.

☐ Verdadero ☐ Falso

### 3.- Gestores de bases de datos orientadas a objetos.

## Caso práctico

Esta mañana, **Juan** ha preparado una serie de sistemas de bases de objetos diferentes, para instalarlos e ir probando algunas de sus características.

Juan le pregunta a **Ana** —¿Sabes qué al contrario de las bases de datos relacionales, los gestores de bases de datos orientados a objetos pueden llegar a ser muy diferentes entre sí?

A lo que **Ana** responde —Sí, en clase estuvimos trabajando con diferentes sistemas, precisamente para dar fe de ello, y..., quiero recordar que había una que tenía el nombre de un pintor francés. ¿Matisse?

—Efectivamente —dice **Juan**—, pero hoy he pensado en ver la base de objetos Db4o, que es la que actualmente utiliza el sistema de trenes español AVE. ¿Lo sabías?



Un **Sistema Gestor de Bases de Datos Orientada a Objetos** (SGBDOO) y en inglés ODBMS, Object Databases Management System) es un software específico, dedicado a servir de **interfaz** entre la base de objetos, el usuario y las aplicaciones que la utilizan. Un SGBDOO incorpora el paradigma de Orientación a Objetos y permite el almacenamiento de objetos en soporte secundario:

- ✓ Por ser SGBD debe incluir mecanismos para optimizar el acceso, gestionar el control de concurrencia, la seguridad y la gestión de usuarios, así como facilitar la consulta y recuperación ante fallos.
- ✓ Por ser OO incorpora características de identidad, encapsulación, herencia, polimorfismo y control de tipos.



[\(link: AD05\\_CONT\\_R07\\_SGBDOO.jpg.\)](#)

Cuando aparecieron las bases de datos orientadas a objetos, un grupo formado por desarrolladores y usuarios de bases de objetos, denominado ODMG (Object-Oriented Database Management Group), propuso un estándar que se conoce como estándar **ODMG-93** y que se ha ido revisando con el tiempo, pero que en realidad **no ha tenido mucho éxito**, aunque es un punto de partida.

Tal y como estarás pensando, la carencia de un estándar real hace difícil el soporte para la **portabilidad** de aplicaciones y su **interoperabilidad**, y es en parte por ello, que a diferencia de las bases de datos relacionales donde hay muchos productos donde elegir, la variedad de sistemas de bases de datos orientadas a objetos es mucho menor. En la actualidad hay diferentes productos de este tipo, tanto con licencia libre como propietaria.

A continuación te indicamos algunos **ejemplos de SGBDOO**:

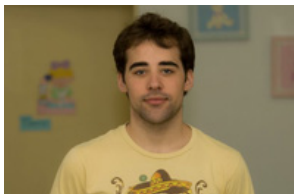
- ✓ **Db4o de Versant**. Es una BDOO Open Source para Java y .NET. Se distribuye bajo licencia **GPL**.
- ✓ **Matisse**. Es un SGBDOO basado en la especificación ODMG, proporciona lenguajes para definición y manipulación de objetos, así como interfaces de programación para C, C++, Eiffel y Java.
- ✓ **ObjectDB**. Es una BDOO que ofrece soporte para Java, C++, y Python entre otros lenguajes. No es un producto libre, aunque ofrecen versiones de prueba durante un periodo determinado.
- ✓ **EyeDB**. Es un SGBDOO basado en la especificación ODMG, proporciona lenguajes para definición y manipulación de objetos, e interfaces de programación para C++ y Java. Se distribuye bajo licencia **GNU** y es software libre.
- ✓ Neodatis, ObjectStore y GemStone. Son otros SGBDOO.

Dada la escasa implantación de los SGBDOO nos centraremos en las posibilidades que nos ofrecen las bases de datos objeto-relacionales.

## 4.- Características de las bases de datos objeto-relacionales.

### Caso práctico

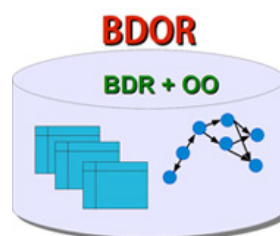
Hoy **Antonio** va a estar todo el día con **Juan y Ana**. Va a aprovechar que hoy comienzan a repasar las posibilidades de las bases de datos objeto-relacionales y como al fin y al cabo son bases de datos relacionales, quiere hacerles unas preguntillas sobre unas dudas que tiene sobre SQL, y de paso empaparse de aquellas características que tienen que ver con la orientación a objetos.



Las BDOR las podemos ver como un híbrido de las BDR y las BDOO que intenta aunar los beneficios de ambos modelos, aunque por descontado, ello suponga renunciar a algunas características de ambos.

Los objetivos que persiguen estas bases de datos son:

- ✓ Mejorar la representación de los datos mediante la orientación a objetos.
- ✓ Simplificar el acceso a datos, manteniendo el sistema relacional.



[.\(link: AD05\\_CONT\\_R39\\_BDOR.jpg.\)](#)

En una BDOR se siguen almacenando tablas en filas y columnas, aunque la estructura de las filas no está restringida a contener escalares o valores atómicos, sino que las columnas pueden almacenar tipos estructurados (tipos compuestos como vectores, conjuntos, etc.) y las tablas pueden ser definidas en función de otras, que es lo que se denomina herencia directa.

Y eso, ¿cómo es posible?

Pues porque internamente tanto las tablas como las columnas son tratados como objetos, esto es, se realiza un mapeo objeto-relacional de manera transparente.

Como consecuencia de esto, aparecen **nuevas características**, entre las que podemos destacar las siguientes:

- ✓ **Tipos definidos por el usuario**. Se pueden crear nuevos tipos de datos definidos por el usuario, y que son compuestos o estructurados, esto es, será posible tener en una columna un atributo multivaluado (un tipo compuesto).
- ✓ **Tipos Objeto**. Posibilidad de creación de objetos como nuevo tipo de dato que permiten relaciones anidadas.
- ✓ **Reusabilidad**. Posibilidad de guardar esos tipos en el gestor de la BDOR, para reutilizarlos en tantas tablas como sea necesario.
- ✓ **Creación de funciones**. Posibilidad de definir funciones y almacenarlas en el gestor. Las funciones pueden modelar el comportamiento de un tipo objeto, en este caso se llaman métodos.
- ✓ **Tablas anidadas**. Se pueden definir columnas como arrays o vectores multidimensionales, tanto de tipos básicos como de tipos estructurados, esto es, se pueden anidar tablas.
- ✓ **Herencia** con subtipos y subtablas.

Estas y otras características de las bases de datos objeto-relacionales vienen recogidas en el estándar SQL 1999 .

### Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.



**En una base de datos objeto-relacional, las columnas de una tabla pueden almacenar tipos estructurados.**

☐ Verdadero ☐ Falso

## 4.1.- El estándar SQL99.

La norma ANSI SQL1999 (abreviadamente, SQL99) extiende el estándar SQL92 de las Bases de Datos Relacionales, y da cabida a nuevas características orientadas a objetos preservando los fundamentos relacionales.



Algunas extensiones que contempla este estándar y que están relacionadas directamente con la orientación a objetos son las siguientes:

### ✓ Extensión de tipos de datos.

- ✓ Nuevos tipos de datos básicos para datos de caracteres de gran tamaño, y datos binarios de gran tamaño (Large Objects)
- ✓ Tipos definidos por el usuario o tipos estructurados.
- ✓ Tipos colección, como los arrays, set, bag y list.
- ✓ Tipos fila y referencia

### ✓ Extensión de la semántica de datos.

- ✓ Procedimientos y funciones definidos por el usuario, y almacenados en el gestor.
- ✓ Un tipo estructurado puede tener métodos definidos sobre él.

Por ejemplo, el siguiente segmento de SQL crea un nuevo tipo de dato, un tipo estructurado de nombre profesor y que incluye en su definición un método, el método `sueldo()`.

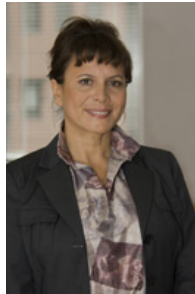
```
CREATE TYPE profesor AS (  
  id INTEGER,  
  nombre VARCHAR (20),  
  sueldo_base DECIMAL (9,2),  
  complementos DECIMAL (9,2),  
  INSTANTIABLE NOT FINAL  
  METHOD sueldo() RETURNS DECIMAL (9,2));  
CREATE METHOD sueldo() FOR profesor  
BEGIN  
  .....  
END;
```

### Caso práctico

**Ada** ha reunido a **Juan** y **Ana** esta mañana, para hablar sobre la marcha del proyecto, y decirles que **María** empezará esta semana a trabajar algunos ratos con ellos. **Ada** les dice —El cliente de la aplicación de mobiliario de diseño nos ha pedido que le diseñemos también un sitio web para su negocio, de manera que **María** se encargará de ese tema —y pregunta—, ¿os habéis decidido ya por algún sistema de bases de datos en particular?

A lo que **Juan** responde. —Esta semana vamos a valorar algunos sistemas objeto-relacionales, aunque casi estamos seguros de que para esta aplicación lo mejor es una base de objetos.

**Ada** sonrío y dice —Bueno, aplicaos en el tema y cuando tengáis la decisión tomada, me lo comunicáis.



Podemos decir que un sistema gestor de bases de datos objeto-relacional (SGBDOR) contiene dos tecnologías; la tecnología relacional y la tecnología de objetos, pero con ciertas restricciones.

A continuación te indicamos algunos ejemplos de **gestores objeto-relacionales**, tanto de código libre como propietario, todos ellos con soporte para Java:

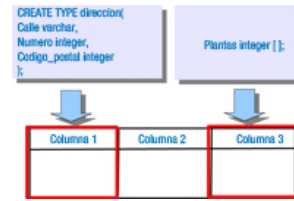
- ✓ De código abierto:
  - ✓ **PostgreSQL**
  - ✓ **Apache Derby**
- ✓ De código propietario
  - ✓ **Oracle**
  - ✓ **First SQL**
  - ✓ **DB2 de IBM**

Las normativas SQL99 y SQL2003 son el estándar base que siguen los SGBDOR en la actualidad, aunque como siempre ocurre, cada gestor incorpora sus propias particularidades y diferencias respecto al estándar. Por ejemplo, en Oracle se pueden usar las dos posibilidades de herencia, de tipos y de tablas, mientras que en PostgreSQL solo se puede usar la herencia entre tablas.

## 5.1.- Características y tipos de objetos

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



([link: AD05\\_CONT\\_R45\\_TDU.jpg](#))

En este apartado estudiaremos la orientación a objetos que proporciona Oracle.

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.

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

### 5.1.1.- Método member

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);
```

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:**

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;
```

## 5.1.2.- Método constructor

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.
);
```

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 inician 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.

### 5.1.3.- Métodos map y order

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.

## 5.2.- 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          NOT NULL      VARCHAR2(35)
DIREC           NOT NULL      DIRECCION
FECHA_NAC       NOT NULL      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 espártalos 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 IN C1 LOOP
    DBMS_OUTPUT.PUT_LINE(I.NOMBRE || ' - Calle: ' || I.DIREC.CALLE);
  END LOOP;
END;
```



## 5.3.- 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 permiten almacenar un conjunto de elementos, y de las tablas anidadas que permiten almacenar en una columna de una tabla otra tabla.

### 5.3.1.- 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:

Creemos 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;
```

### 5.3.2.- 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                                VARCHA2(25)  
CIUDAD                               VARCHA2(20)  
CODIGO_POST                           NUMBER(5)  
  
METHOD  
-----  
MEMBER PROCEDURE SET_CALLE  
Nombre de Argumento                 Tipo                               E/S                               Por Defecto  
-----  
C                                   VARCHA2                           IN  
  
METHOD  
-----  
MEMBER FUNCTION GET_CALLE RETURNS VARCHA2  
  
SQL> DESC EJEMPLO_TABLA_ANIDADA ;  
Nombre                               Nulo                               Tipo  
-----  
ID                                   NUMBER(2)  
APELLIDOS                           VARCHA2(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.

## 5.4.- 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';
```

## 5.5.- 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;
```



### Caso práctico

—¡Por fin ha llegado el viernes! —dice **Ana** a **Juan**, y continúa— después de estas dos últimas semanas revisando sistemas objeto-relacionales ¡he terminado agotada!, pero el esfuerzo me ha compensado. Ahora tengo mucho más claro, las diferencias entre una base de objetos y una objeto-relacional. Pero... —se queda pensativa por un instante y entonces continúa— las transacciones ¿se gestionan de manera similar en ambos tipos de sistemas?

**Juan** le responde, —justo es eso lo que nos queda por ver, **Ana**. ¿Empezamos hoy o lo dejamos para el lunes?

**Ana** sonríe y le dice —Creo que **Ada** viene para acá, ¿no oyes sus pasos?



Como en cualquier otro Sistema de Bases de datos, en un Sistema de bases de objetos u objeto relacional, **una transacción** es un conjunto de sentencias que se ejecutan formando una unidad de trabajo, esto es, en forma indivisible o atómica, o se ejecutan todas o no se ejecuta ninguna.

Mediante la gestión de transacciones, los sistemas gestores proporcionan un acceso concurrente a los datos almacenados, mantienen la integridad y seguridad de los datos, y proporcionan un mecanismo de recuperación de la base de datos ante fallos.

Un ejemplo habitual para motivar la necesidad de transacciones es el traspaso de una cantidad de dinero (digamos 10000€) entre dos cuentas bancarias. Normalmente se realiza mediante dos operaciones distintas, una en la que se decrementa el saldo de la cuenta origen y otra en la que incrementamos el saldo de la cuenta destino.

Para garantizar la integridad del sistema (es decir, para que no aparezca o desaparezca dinero), las dos operaciones tienen que completarse por completo, o anularse íntegramente en caso de que una de ellas falle.

Las transacciones deben cumplir el criterio ACID.

- ✓ **Atomicidad**. Se deben cumplir todas las operaciones de la transacción o no se cumple ninguna; no puede quedar a medias.
- ✓ **Consistencia**. La transacción solo termina si la base de datos queda en un estado consistente.
- ✓ **Isolation** (Aislamiento). Las transacciones sobre la misma información deben ser independientes, para que no interfieran sus operaciones y no se produzca ningún tipo de error.
- ✓ **Durabilidad**. Cuando la transacción termina el resultado de la misma perdura, y no se puede deshacer aunque falle el sistema.

Algunos sistemas proporcionan también **puntos de salvaguarda** (savepoints) que permiten descartar selectivamente partes de la transacción, justo antes de acometer el resto. Así, después de definir un punto como punto de salvaguarda, puede retrocederse al mismo. Entonces se descartan todos los cambios hechos por la transacción después del punto de salvaguarda, pero se mantienen todos los anteriores.

Originariamente, los puntos de salvaguarda fueron una aportación del estándar SQL99 de las Bases de Datos Objeto-Relacionales. Pero con el tiempo, se han ido incorporando también a muchas Bases de Datos Relaciones como MySQL (al menos cuando se utiliza la tecnología de almacenamiento InnoDB).

## 6.1.- Transacciones en una base objeto-relacional.

Los SGBDOR gestionan transacciones mediante las sentencias COMMIT (confirmar transacción) y ROLLBACK (deshacer transacción).

JDBC permite agrupar instrucciones SQL en una sola transacción. Así, podemos asegurar las propiedades ACID usando las facilidades transaccionales del JDBC.

El control de la transacción lo realiza el objeto `Connection`. Cuando se crea una conexión, por defecto es en modo `AUTO COMMIT= TRUE`. Esto significa que cada instrucción individual SQL se trata como una transacción en sí misma, y se confirmará en cuanto que la ejecución termine.

Por tanto, si queremos que un grupo de sentencias se ejecuten como una transacción, tras abrir una conexión `Connection conn`; habrá que:

- ✓ Poner `autocommit=false` de la siguiente manera:

```
if (conn.getAutoCommit() )  
    conn.setAutoCommit( false );
```

- ✓ Mediante un bloque try-catch controlar si se deshace `conn.rollback` o confirma `conn.commit` la transacción iniciada con esa conexión.

# Anexo I.- Estándar ODMG-93 u ODMG.

[\(link:.\)](#)

[\(link:.\)](#)

El **estándar ODMG** (Object Database Management Group) trata de estandarizar conceptos fundamentales de los Sistemas Gestores de Bases de Datos Orientados a Objetos (SGBDOO) e intenta definir un **SGBDOO como un sistema que integra las capacidades de las bases de datos con las capacidades de los lenguajes de programación orientados a objetos**, de manera que los objetos de la base de datos aparezcan como objetos del lenguaje de programación.

Fué desarrollado entre los años 1993 y 1994 por representantes de un amplio conjunto de empresas relacionadas con el desarrollo de software y sistemas orientados a objetos.

## 1. Arquitectura del estándar ODMG

La arquitectura propuesta por ODMG consta de:

- ✓ Un **modelo de objetos** que permite que tanto los diseños, como las implementaciones, sean portables entre los sistemas que lo soportan.
- ✓ Un **sistema de gestión** que soporta un lenguaje de bases de datos orientado a objetos, con una sintaxis similar a un lenguaje de programación también orientado a objetos.
- ✓ Un **lenguaje de base de datos** que es especificado mediante:
  - ✓ Un Lenguaje de Definición de Objetos (ODL)
  - ✓ Un Lenguaje de Manipulación de Objetos (OML)
  - ✓ Un Lenguaje de Consulta (OQL)

siendo todos ellos portables a otros sistemas con el fin de conseguir la portabilidad de la aplicación completa.

- ✓ Enlaces con lenguajes Orientados a Objetos como C++, Java, Smaltalk.

El **modelo de objeto ODMG** es el modelo de datos en el que están basados el ODL y el OQL. Este modelo de objeto proporciona los tipos de datos, los constructores de tipos y otros conceptos que pueden utilizarse en el ODL para especificar el esquema de la base de datos de objetos.

Vamos a destacar algunas de las **características más relevantes** del estándar ODMG:








- ✓ Las primitivas básicas de modelado son los **objetos** y los **literales**.
- ✓ Un objeto tiene un **Identificador de Objeto** (OID) y un estado (valor actual) que puede cambiar y tener una estructura compleja. Un literal no tiene OID, pero si un valor actual, que es constante.
- ✓ El estado está definido por los valores que el objeto toma para un conjunto de propiedades. Una propiedad puede ser:
  - ✓ Un atributo del objeto.
  - ✓ Una interrelación entre el objeto y otro u otros objetos.
- ✓ Objetos y literales están organizados en **tipos**. Todos los objetos y literales de un mismo tipo tienen un comportamiento y estado común.
- ✓ Un objeto queda descrito por cuatro características: identificador, nombre, tiempo de vida y estructura.
- ✓ Los tipos de objetos se descomponen en atómicos, colecciones y tipos estructurados.
  - ✓ **Tipos atómicos** o básicos: constan de un único elemento o valor, como un entero.
  - ✓ **Tipos estructurados**: compuestos por un número fijo de elementos que pueden ser de distinto tipo, como por ejemplo una fecha.
  - ✓ **Tipos colección**: número variable de elementos del mismo tipo. Entre ellos:
    - ✓ Set: grupo desordenado de elementos y sin duplicados.

- ✓ Bag: grupo desordenado de elementos que permite duplicados.
- ✓ List: grupo ordenado de elementos que permite duplicados.
- ✓ Array : grupo ordenado de elemntos que permite el acceso por posición.

Algunos fabricantes sólo ofrecen vinculaciones de lenguajes específicos, sin ofrecer capacidades completas de ODL y OQL.

## Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	<p>Autoría: Agustín Díaz.</p> <p>Licencia: CC-BY-SA.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/yonmacklein/22105978/">http://www.flickr.com/photos/yonmacklein/22105978/</a></p>		<p>Autoría: Pablo Sanz Almogera.</p> <p>Licencia: CC BY-NC-SA.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/pablosanz">http://www.flickr.com/photos/pablosanz</a></p>
	<p>Autoría: Luis Pérez.</p> <p>Licencia: CC BY-NC-SA.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/luipermom/2934628860/">http://www.flickr.com/photos/luipermom/2934628860/</a></p>		<p>Autoría: RonLD.</p> <p>Licencia: CC-BY.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/rld/12045">http://www.flickr.com/photos/rld/12045</a></p>
	<p>Autoría: Jeff Kubina.</p> <p>Licencia: CC BY-SA.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/kubina/912714753">http://www.flickr.com/photos/kubina/912714753</a></p>		<p>Autoría: Sperc.</p> <p>Licencia: CC BY-NC.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/ericfarkas">http://www.flickr.com/photos/ericfarkas</a></p>
	<p>Autoría: Olga Berrios.</p> <p>Licencia: CC BY.</p> <p>Procedencia:  <a href="http://www.flickr.com/photos/ofernandezberrios/368893337/">http://www.flickr.com/photos/ofernandezberrios/368893337/</a></p>		

