

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

## **Bases de datos orientadas a objetos**

## Definición de las bases de datos orientadas a objetos

Relacionamos el concepto de bases de datos orientadas a objetos, cuando las técnicas de bases de datos se combinan con objetos. El resultado de esto es un **sistema gestor orientado a objetos: ODBMS (Object Data Base Management System)**. Hoy en día, realmente existe un aumento del desarrollo de aplicaciones orientadas objetos, facilitando, por tanto, la evolución de los sistemas gestores de base de datos orientados a objetos (OODBMS). Esto proporciona un gran valor para los desarrolladores que trabajan con este tipo de aplicaciones. Es evidente que estos profesionales tendrán cierta facilidad para realizar el desarrollo de la **aplicación orientada a objetos**, almacenarla en una **base de datos realizada** en el mismo paradigma, y **replicar o modificar objetos** existentes para crear nuevos **objetos dentro de este sistema** OODMBS.

El modelo de datos orientado a objetos (OODM) son modelos lógicos que capturan la semántica de los objetos de la aplicación que va unida o relacionada. Realmente, son modelos que van en consonancia con el modelo de la aplicación. Los OODMs **implementan modelos conceptuales** directamente y pueden representar **complejidades** que van **más allá** de los sistemas de **bases de datos relacionales**. Por este motivo, han heredado muchos de los conceptos que fueron pensados e implementados en los lenguajes de programación orientados a objetos.

Realmente, una **base de datos orientada a objetos** no es más que:

→ una **colección de objetos** definidos por un **modelo orientado a objetos**.

Este tipo de bases de datos pueden **extender la existencia de los objetos** para que se **almacenen indefinidamente**.

Por lo tanto, los objetos son almacenados más allá de la finalización de la aplicación con la que estamos trabajando.

Pueden ser **recuperados** más tarde y **compartidos** por otras aplicaciones.

## Características de las bases de datos orientadas a objetos

- Mantiene una **relación directa** entre el **mundo real y los objetos** de la base de datos. Los **objetos no pierden ni su identidad ni su integridad**.
- Proporciona un **identificador de objeto, generado por el sistema** para cada objeto para que un objeto pueda identificarse fácilmente.
- Son **fáciles de extender** y de **añadir nuevos tipos de datos y operaciones** que se realizarán en ellos.
- Proporciona **encapsulación**. La representación de la información y las implementaciones de los métodos **ocultan entidades externas**.
- También proporciona propiedades de **herencia**, mediante la cual un objeto hereda las propiedades de otros objetos.

### Herencia de objetos

La herencia es una característica muy potente que nos permitirá **definir** una **clase** teniendo en cuenta, y tomando como **base, otra** ya existente. Una de las grandes ventajas de la herencia es la **reutilización de código**.

Los ODBMS proporcionan propiedades de herencia, permitiendo que un objeto **herede las propiedades de otros objetos**.

Los **sistemas relacionales** han implementado estrategias para modelar la herencia, como **tablas separadas o claves foráneas**, pero **no de manera tan inherente** como en los ODBMS.

### Conceptos asociados a los objetos:

- **Atributos:**

Son las **características** que suelen describir los objetos. También conocidos como **variables de instancia**. Cuando los atributos son **asignados a valores** en un momento dado, se asume que el objeto está en un **estado determinado** en ese momento.

- **Objeto:**

Un objeto es una **representación abstracta** de una **entidad** del mundo real, la cual tiene un **identificador único, propiedades embebidas** y la capacidad de **interactuar** con otros objetos por sí mismo.

- **Identidad:**

La identidad es un **identificador externo** (el ID del objeto) que **se mantiene** por cada objeto. El ID del objeto es **asignado por el sistema** cuando el **objeto es creado y no puede ser cambiado**.

Es distinto a las **bases de datos relacionales**, ya que, por ejemplo, este **ID** está **almacenado en el interior** del objeto y, además, se usa para **identificarlo**.

#### **Identificador de Objeto Generado por el Sistema:**

Los ODBMS proporcionan un **identificador único** generado por el sistema para cada objeto, facilitando la identificación única de los objetos.

Los **sistemas relacionales** también proporcionan identificadores únicos (**claves primarias**), pero la generación y gestión de estas claves pueden requerir una **configuración más explícita**.

*A diferencia de las **bases de datos relacionales**, donde los identificadores suelen ser **manejados internamente por el sistema** y **no están accesibles directamente al usuario**, en las **bases de datos orientadas a objetos**, el ID del objeto puede estar almacenado **dentro del objeto mismo** y ser **accesible para los programadores**.*

#### **Inmutabilidad del ID del Objeto:**

*Un aspecto importante es que una vez que se asigna un ID a un objeto, este ID generalmente **no puede ser cambiado** a lo largo de la vida del objeto. Permanece **constante** desde la creación hasta la eliminación del objeto.*

*En resumen, en **bases de datos orientadas a objetos**, la identidad se gestiona mediante un **identificador único** asignado a cada objeto, y este identificador a menudo está almacenado dentro del objeto mismo, siendo **inmutable** a lo largo de la vida del objeto. Este enfoque es diferente de las **bases de datos relacionales** donde los identificadores suelen ser **manejados internamente por el sistema**.*

## Sistemas gestores de bases de datos orientados a objetos

Un sistema gestor de base de datos orientado a objetos se constituye, básicamente, con un sistema gestor de almacenamiento de datos que soporta el modelado y la creación de los datos como objetos. Permite la **conurrencia** y la **recuperación**. Para los consumidores de bases de datos relacionales, significa **olvidarse de** la traducción de **filas** y **columnas**, y, por lo tanto, manipular **directamente con objetos**.

Un sistema gestor de base de datos posee una serie de **datos relacionados entre sí** y una aplicación o **aplicaciones** rodeando dicha base de datos que tendrá **acceso a ella**.

### Características de un sistema gestor de base de datos genérico:

- **Conurrencia.**
- **Persistencia.**
- **Recuperación de errores.**
- Gestión de **almacenamiento**.
- **Consultas.**

### Característica de un sistema gestor de base de datos orientado a objetos, además:

- **Abstracción.**
- **Modularidad.**
- **Jerarquía.**
- **Encapsulación.**
- **Tipología** de objetos.

## ObjectDB

Se puede usar para realizar un **ágil manejo de bases de datos** orientadas a objetos.

Posee una gran variedad de **funcionalidades**, pero habría que destacar, para este caso:

- **Data base explorer:**

Es la herramienta visual de la aplicación donde podremos realizar las **consultas**, **visualizar** los objetos y **editar** el contenido.

- **Database Doctor:** Realiza una serie de **diagnósticos** en relación a posibles problemas de la base de datos.

- **Replication:** Realiza **copias maestro-esclavo**.

- **Online Backup:** Podremos realizar una **copia de seguridad** a través de una **consulta** en un **EntityManager**.

## Lenguaje de consultas para objetos

El lenguaje de consulta de objetos u **Object Query Language (OQL)** es un **lenguaje declarativo** de **tipología SQL** que nos facilitará realizar **consultas** de modo efectivo en bases de datos orientadas a objetos y a estructuras de los mismos.

Este lenguaje **no contiene primitivas** que se ocupen de **modificar el estado** de dichos **objetos**, ya que este tipo de modificaciones se realizarán a través de **métodos que poseen los objetos**.

Partimos de una **estructura básica** de **SELECT, FROM, WHERE** como en el conocido lenguaje SQL. Podemos ver el siguiente ejemplo:

### Ejemplo OQL

```
SELECT c.name  
FROM c in customer  
WHERE c.department = 'e-commerce';
```

### Agregación y asociación

La agregación es un tipo específico de asociación.

**Asociación:** Cuando **borramos** un objeto que forma parte de una asociación, el resto de objetos **relacionados continúan existiendo**.

**Agregación:** Cuando introducimos o **borramos** un objeto que forma parte de una agregación es igual a insertar o **eliminar** el resto de sus **componentes relacionados**. De esta forma, un objeto que pertenece a otro **no puede ser introducido o borrado de forma aislada** en la base de datos.

### Especialización, generalización y herencia

Definimos una **clase** para organizar una serie de **objetos parecidos**.

En algunos casos, los **objetos de una clase** pueden ser **organizados de nuevo** formando ciertas **agrupaciones** que pueden ser **relevantes** para la base de datos.

Podemos ver, en el siguiente ejemplo, que cada una de las agrupaciones formadas son subclases de la clase VEHICULO, y que VEHICULO coge el mandato de superclase del resto de clases. Con esta relación, definimos el concepto de **generalización o especialización**:

*Un ciclomotor es un vehículo.*

*Un coche es un vehículo.*

*Un autobús es un vehículo.*

*Cada subclase de vehículo es una especialización de vehículo, y vehículo es una generalización de sus subclases ciclomotor, coche y autobús.*

## Consultas de un objeto

Vamos a ver distintas formas de consultar el objeto CUSTOMER que vimos en ejemplo OQL anterior.

Las **variables** llamadas **de tipo «iterador»** se pueden **nombrar de 3 formas distintas**:

- **C in Customer.**
- **Customer C.**
- **Customer as C.**

El resultado de este tipo de query podría ser **válido** si forma parte de la **tipología definida previamente en el modelo**.

Por este motivo, **no es obligatorio** que posea la cláusula **SELECT**, ya que, simplemente **nombrando** cualquier **objeto**, se **devolverán** todas sus **existencias**. Es decir:

- **Customer;** Devolverá una colección de todos los objetos de tipo Customer que existan en la base de datos. De la misma forma, si existiera un objeto concreto PremiumCustomer, se realizaría la siguiente consulta:
  - **PremiumCustomer;** Obtendremos el resultado de **ese tipo específico de Customer**.

Por lo general, una **consulta OQL** comienza con el **nombre del objeto** persistente y, a continuación, se le añade uno, varios **atributos** o ninguno, **mediante un punto**.

Veamos algún **ejemplo**:

**customer.empresa:** devolvería un objeto de tipo Empresa, que estaría **vinculado ese Customer**.

**customer.empresa.nombre:** se accede al atributo de la empresa llamado nombre. Por lo tanto, nos devolvería un objeto de tipo **String con el nombre de la empresa**.

**customer.empresas\_asociadas:** nos devolvería un Set<Empresa>, una colección de tipo Empresa donde podríamos ver **todas las empresas** que ese cliente tuviera asociadas.



## Ventajas en la utilización de las bases de datos orientadas a objetos

Estas son algunas de las ventajas y desventajas que presenta desarrollar nuestra aplicación orientada a objetos:

- Dispondremos de una capacidad mayor de realizar el **modelado**. Esto podemos considerarlo debido a:
  - Dentro de los objetos podremos **encapsular** tanto **comportamientos** como **estados**.
  - Las **relaciones** de un objeto pueden ser almacenadas **en su interior**.
  - Al agruparse, los objetos forman **objetos complejos**. Es el concepto que denominamos como **herencia**.
- También dispondremos de una **flexibilidad importante**. Esto se debe a:
  - Podremos construir **nuevos objetos** con tipologías nuevas, partiendo de los que ya tenemos.
  - Se **reduce la redundancia**, ya que podremos aunar características o propiedades de distintas clases y agruparlas en **superclases**.
  - Las **clases** existentes u **objetos** son **reusables**. Lo cual influye directamente en el **tiempo de desarrollo**.
- Por medio de la **intuición**, podremos realizar de forma práctica algunas de las consultas, ya que es un **lenguaje expresivo**. Es realmente fácil navegar entre objetos y sus herencias, ya que es un **acceso navegacional**.
- **Rendimiento** muy competitivo. Algunos autores han comparado rendimientos de bases de datos orientadas a objetos y bases de datos relacionales. **En aplicaciones orientadas a objetos**, el rendimiento de la base de datos orientada a objetos es **superior**.

## Desventajas en la utilización de las bases de datos orientadas a objetos

- La **falta** de un **modelo de datos universal**: No existe dicho modelo de datos universalmente aceptado y a la mayoría de los sistemas gestores de bases de datos orientados a objetos les falta una buena **base teórica**.
- **Falta de experiencia**: Es evidente que no es comparable, en madurez, con los sistemas de base de datos relacionales.
- **Falta de estándares**: Realmente, no existen estándares definidos como tal.
- La **competencia**: Tal y como hemos comentado anteriormente **respecto a los sistemas relacionales**, incluso los objeto-relacionales poseen una gran competencia, ya que dichos sistemas tienen un gran **estándar aprobado**, como SQL, y una base teórica sólida. Además, los sistemas relacionales disponen de un **entramado de aplicaciones** de soporte alrededor de ellas, tanto para usuarios como para programadores.
- La **encapsulación** es casi una forma **obligada** de realizar consultas: El desarrollo de los objetos de forma encapsulada es prácticamente una obligación, ya que es la **forma que accederemos** a futuro mediante el sistema de consultas.
- En relación a la **teoría matemática**: Podemos decir que el modelo de objetos aún **no posee una aprobación**.

## Programación de aplicaciones con acceso a bases de datos orientadas a objetos.

### Diseño de una API (Application Programming Interface)

**API (Application Programming Interface)** define una serie de **especificaciones**, de reglas a cumplir, para consumir ciertas **funcionalidades** de un **sistema externo** determinado.

Hoy en día, el concepto API está muy orientado a las **aplicaciones web** con patrón de diseño **REST**.

En esta arquitectura, se definen una serie de **Endpoints** o puertas de entrada al **código back** de la aplicación con la que hay que conectar y de la que hay que **obtener las funcionalidades** o servicios.

Estas funcionalidades han debido ser **previamente definidas y analizadas**, y deben cubrir la totalidad de la **lógica** que un sistema de **back-end puede devolver** a un front-end.

Si, por **ejemplo**, estamos ante una aplicación de gestión de clientes donde alguna de las **funcionalidades** de la propia aplicación son **crear** clientes, **modificar** clientes y **eliminar**, en la **API** o interfaz de la aplicación habrá **3 EndPoints**, que serán la **puerta principal al código back-end**.

Los EndPoints son, básicamente, **métodos** definidos como entrada para **realizar** dichas **funcionalidades vinculados por un path** que los denomina.

Este sería el caso de una API en tecnología o patrón de diseño REST.

Con respecto a las **bases de datos orientadas a objetos**, debemos ponernos en contexto sobre **qué disponemos y qué necesitamos**.

En nuestro caso, y atendiendo a la temática que concierne esta unidad, **disponemos** de una **aplicación** que en su tramo final **accede a una base de datos orientada a objetos**. En ese punto, sería de gran interés analizar y diseñar una API en la **capa DAO** (Data Access Object), capa de la aplicación donde **accedemos a datos**.

Sería recomendable (y signo de robustez y orden) diseñar una **interfaz** con una serie de **métodos** cuyo objetivo fuera poblar de todas las **respuestas necesarias** en cuanto a **datos** se refiere. Es decir, definir los llamados **Endpoints** cuyo **nombre** dan respuesta al objeto u **objetos que se van a devolver**.

Ejemplo EndPoints:

@GetMapping y @PostMapping serían las puertas de entrada.

```
@RestController
public class ReservationController {

    @Autowired
    private ReservationService reservationService;

    /**
     * Use this endPoint in order to get all the available (FREE) Reservation
     * @return List of ReservationDto
     */
    @GetMapping("/reservations/available")
    List<ReservationDto> getAvailableReservations() { return reservationService.getAvailableReservations(); }

    /**
     * Use this endPoint in order to update one Reservation
     * @param id reservation identifier
     * @param status reservation status: FREE/BLOCKED/CONFIRMED
     * @return ReservationDto Object
     */
    @PostMapping("/reservations/edit/{id}/{status}")
    ReservationDto editReservation(@PathVariable Long id, @PathVariable String status) {

        return reservationService.updateReservation(id, status);
    }
}
```

### Gestión de transacciones en las bases de datos orientadas a objetos

Realmente no vamos a observar gran diferencia en cuanto al concepto de transacción en una base de datos SQL.

Una **transacción** seguirá siendo un conjunto de sentencias que formaran una unidad de trabajo, seguirá manteniendo características de **atomicidad**, **consistencia**, **aislamiento**... solo que en el contexto de las bases de datos orientadas a objetos, dichas transacciones serán **realizadas mediante el uso de objetos**.

## Ejemplo de uso de lenguaje OQL

La siguiente base de datos está formada por **objetos**. Algunos de ellos y sus **atributos** son:

- **Alojamiento:**
  - Object Pared.
  - Object Suelo.
  - Object Persiana.
- **Pared:**
  - String tipoPared.
  - String color.
- **Suelo:**
  - Int extension.
- **Persiana:**
  - Int ancho.
  - Int largo.
  - String color.

Vamos a seleccionar todos aquellos **alojamientos** cuyo **suelo sea superior a 50** y coincidan con el **color de pared blanco**:

Selección de alojamiento, ejemplo OQL:

```
SELECT *  
FROM a in alojamiento  
WHERE a.suelo.extension > 50 AND a.pared.color='blanco'
```

Ahora vamos a seleccionar aquellos **alojamientos** donde la **persiana** tenga un **largo inferior a 100**, el **color de la persiana sea «metalico»** y el tipo de la **pared** sea «**gotelé**».

Especificación de persiana y pared:

```
SELECT *  
FROM a in alojamiento  
WHERE a.persiana.largo < 100 AND a.persiana.color='metalico' AND a.pared.tipoPared='gotele'
```



Un **ingeniero** es un objeto **independiente**, pero que **forma parte del** objeto **empleado**, que es otro **objeto**, y que, a su vez, **forma parte del** objeto **persona**, que **engloba a todos ellos**.

Para acceder al atributo **nombre** de un empleado de tipo **ingeniero**, valdría con:

**Persona.empleado.ingeniero.nombre**

De esta forma, devolveríamos el atributo nombre, que estaría definido como un String.

Cabe destacar que todos y cada uno de los **objetos** por los que se ha pasado tendrán **diferentes atributos**.