

TEMA 4. BASES DE DATOS OBJETO-RELACIONALES Y ORIENTADAS A OBJETOS

2º DAM

Mª del Mar Pérez Roperó

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 (POO), 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.

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 e 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.

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** (conjunto de valores que toman sus propiedades (atributos y relaciones) como el **comportamiento** (conjunto de operaciones que se pueden realizar sobre un objeto) 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:

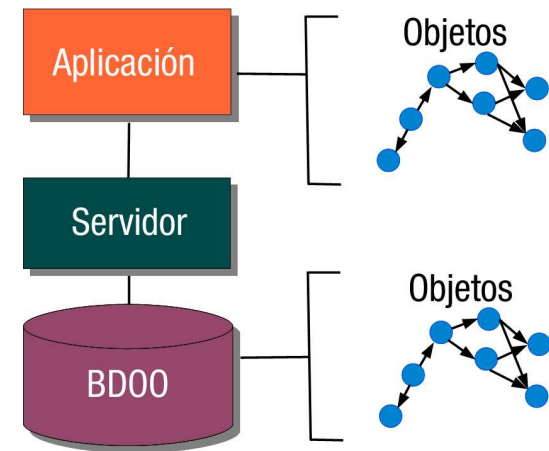
- **Soportan las características propias de la Orientación a Objetos** como agregación, encapsulamiento, polimorfismo y herencia. La herencia se mantiene en la propia base de datos.
- **Identificador de objeto (OID).** Cada objeto tiene un identificador, generado por el sistema, que es único para cada 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.

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.



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:

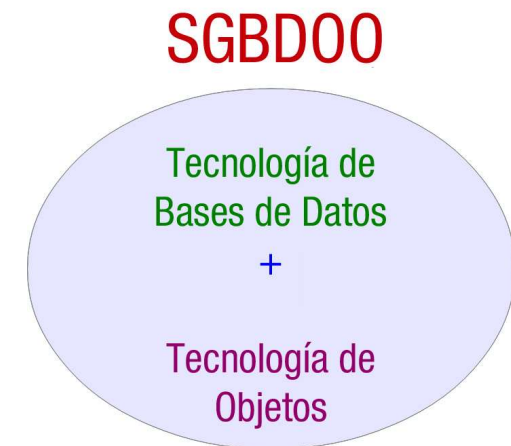
- **Carencia de un modelo de datos universal.** No hay ningún modelo de datos aceptado universalmente, y la mayor parte de los modelos carecen de una base teórica.
- **Falta de estándares.** Existe una carencia de estándares general para los sistemas de BDOO.
- **Complejidad.** La estructura de una BDOO 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.

3. Gestores de bases de datos orientadas a objetos.

Un **Sistema Gestor de Bases de Datos Orientada a Objetos** (SGBDOO) y en inglés ODBMS, Object Databas 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.

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.



¿Qué estrategias o enfoques se siguen para el desarrollo de SGBDOO? Básicamente, las siguientes:

- Ampliar un lenguaje de programación OO existente con capacidades de BD (Ejemplo: GemStone).
- Proporcionar bibliotecas de clases con las capacidades tradicionales de las bases de datos, como persistencia, transacciones, concurrencia, etc., (Ejemplo: ObjectStore y Versant).
- Ampliar un lenguaje de BD con capacidades OO, caso de SQL 2003 y Object SQL (OQL, propuesto por ODMG).

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 SGBOO**:

- **Db4o de Versant.** Es una BDOO Open Source para Java y .NET. Se distribuye bajo licencia GPL. Actualmente es un producto discontinuado, por lo que no se recomienda su uso.
- **Matisse.** Es un SGBOO basado en la especificación ODMG, proporciona lenguajes para definición y manipulación de objetos, así como interfaces de programación para C, C++, Eiffel y Java. Desde 2022 no está disponible, por lo que no se recomienda su uso.
- **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 SGBOO 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.

3.1. Objetos simples y objetos estructurados.

En las BDOO los objetos se encuentran interrelacionados por referencias entre ellos de manera similar a como los objetos se referencian entre sí en memoria.

Un **objeto de tipo simple** u objeto simple es aquel que no contiene a otros objetos y por tanto posee una estructura de un solo nivel de profundidad en este sentido.

Un **objeto de tipo estructurado** u objeto estructurado incluye entre sus componentes a otros objetos y se define aplicando los constructores de tipos disponibles por el SGBDOO recursivamente a varios niveles de profundidad.

Objeto Simple

```
Class Empleado{  
String nif;  
String nombre;  
Double sueldo;  
.....  
}
```

Objeto Estructurado

```
Class Oficina  
String codigo;  
String dirección;  
Empleado jefe;  
.....  
}
```

Entre un objeto y sus componentes de cada nivel, existen dos tipos de referencia:

- **Referencia de propiedad.** Se aplica cuando los componentes de un objeto se encapsulan dentro del propio objeto y se consideran, por tanto, parte de ese objeto. **Relación es-parte-de.** No necesitan tener identificadores de objeto y sólo los métodos de ese objeto pueden acceder a ellos. Desaparecen si el propio objeto se elimina.
- **Referencia de asociación.** Se aplica cuando entre los componentes del objeto estructurado existen objetos independientes, pero es posible hacer referencia a ellos desde el objeto estructurado. **Relación está-asociado-con.** Cuando un objeto estructurado tiene que acceder a sus componentes referenciados, lo hace invocando los métodos apropiados de los componentes, ya que no están encapsulados dentro del objeto estructurado.

Por ejemplo, si observas la figura de la página anterior, en un objeto tipo Oficina los componentes **código** y **dirección** son parte del objeto, mientras que el componente **jefe**, es un objeto independiente que está asociado con el objeto oficina.

Entonces, ¿las referencias de asociación son como las relaciones del modelo relacional? Así es:

- La referencia de asociación representa las relaciones o interrelaciones entre objetos independientes, dando la posibilidad de que los objetos puedan detectarse mutuamente en una o dos direcciones, (lo que en el modelo relacional representamos mediante claves ajenas o foráneas que provienen de relaciones uno a uno, uno a muchos y muchos a muchos).
- Una relación uno a muchos se representa mediante un objeto tipo colección (**List**, **Set**, etc.). En una BDOO la colección se maneja como cualquier otro objeto (aunque potencialmente profundo) y normalmente se podrá recuperar y almacenar el objeto padre junto con la colección asociada y su contenido en una sola llamada.

Además, un objeto miembro referenciado puede ser referenciado por más de un objeto estructurado y, no se elimina automáticamente cuando se elimina el objeto del nivel superior.

3.2. Instalación del gestor de objetos ObjectDB.

ObjectDB es un software multiplataforma y se puede utilizar en varios sistemas operativos con JavaSE5 o superior. Se puede integrar en aplicaciones web JavaEE y Spring y desplegado en contenedores de servlets, así como en servidores de aplicaciones JavaEE.

Se puede utilizar en modo cliente-servidor y en modo embebido. Es la primera base de datos orientada a objetos con soporte integrado para las dos API de bases de datos Java estándar: API de persistencia de Java (JPA - Java Persistence API) y Objetos de datos Java (JDO - Java Data Objects).

Algunas **características importantes de ObjectDB** y el uso de la **API estándar JPA/JDO** son las siguientes:

- El modelo de clases es el propio esquema de la base de datos, por lo que se elimina el proceso de diseño, implementación y mantenimiento de la base de datos.
- Puede trabajar como **base de datos embebida**, lo que significa que se puede distribuir con la aplicación, y solo la aplicación que lanza la base de datos embebida puede acceder a ella, siendo ésta invisible para el usuario final.
- Evita la dependencia del proveedor: cambiar a otro SGBDOO es fácil.

La instalación de ObjectDB consistirá en:

- Instalar el motor de base de datos, que son las clases necesarias para hacer que funcione la API en toda su extensión.
- Instalar alguna aplicación para visualizar los datos con los que se está trabajando. Esto último es necesario, pues en otro caso se trabajaría a ciegas con los datos.

En el siguiente enlace hay un tutorial con los pasos a seguir para la descarga del software ObjectDB y realizar su instalación e integración con el IDE NetBeans.

[Instalación y uso de ObjectDB con NetBeans](https://www.objectdb.com/tutorial/jpa/netbeans) (<https://www.objectdb.com/tutorial/jpa/netbeans>)

Si por el contrario queremos instalarlo en Eclipse, utilizaremos este manual:

[Instalación y uso de ObjectDB con Eclipse](https://www.objectdb.com/tutorial/jpa/eclipse) (<https://www.objectdb.com/tutorial/jpa/eclipse>)

Y si lo que queremos es añadirlo como dependencia de Maven, lo podemos ver en este enlace [aquí \(en el apartado ObjectDB with Maven\)](https://www.objectdb.com/download) (<https://www.objectdb.com/download>)

4. El API de la base de objetos.

Todos los SGBDOO, independientemente de su estrategia de diseño, proporcionan un API (Interfaz de Programación de Aplicaciones), más o menos extenso, disponible para ciertos lenguajes OO. En el caso de ObjectDB utiliza JPA y JDO por lo que no es necesario utilizar paquetes propios.

4.1. Apertura y cierre de conexiones.

En general, la conexión de una aplicación Java con una base de objetos se podrá realizar vía:

- JDBC.
- El API proporcionado por el propio gestor de objetos.



En el caso de ObjectDB, al utilizar JPA, una conexión a una base de datos está representada por la interfaz **EntityManager**. Por lo tanto, para manipular una base de datos ObjectDB necesitamos una instancia de EntityManager. Las operaciones que modifican el contenido de la base de datos también requieren una instancia de EntityManagerTransaction.

Obtener una EntityManagerFactory

Obtener una instancia de EntityManager consta de dos pasos:

Primero necesitamos obtener una instancia de EntityManagerFactory que represente la base de datos relevante.

Luego podemos usar esa instancia de fábrica para obtener una instancia de EntityManager.

JPA requiere la definición de una unidad de persistencia en un archivo XML para poder generar un EntityManagerFactory. Pero al utilizar ObjectDB, podemos definir una unidad de persistencia estándar en un archivo XML o simplemente podemos proporcionar la ruta del archivo de la base de datos de ObjectDB:

```
EntityManagerFactory fem = Persistencia.createEntityManagerFactory(  
    "objectdb:$objectdb/db/points.odb");
```

El método estático `createEntityManagerFactory` espera un nombre de unidad de persistencia como argumento, pero cuando se utiliza ObjectDB, también se acepta cualquier ruta de archivo de base de datos válida (absoluta o relativa). ObjectDB considera que cualquier cadena que comience con el prefijo `objectdb:` o termine con `.odb` o `.objectdb` es una URL de base de datos en lugar de un nombre de unidad de persistencia.

La variable `$objectdb` representa el directorio de inicio de ObjectDB (de forma predeterminada, el directorio en el que está instalado ObjectDB). Si aún no existe ningún archivo de base de datos en la ruta indicada, ObjectDB intentará crear uno.

`EntityManagerFactory` también se usa para cerrar la base de datos una vez que terminamos de usarla:

```
emf.close();
```

Obtener un EntityManager

Una vez que se tiene el EntityManagerFactory podemos obtener fácilmente una instancia de EntityManager:

```
EntityManager em = emf.createEntityManager();
```

La instancia de EntityManager representa una conexión a la base de datos. Cuando se utiliza JPA, cada operación en una base de datos está asociada con un EntityManager. Además, en una aplicación multiproceso, cada subproceso suele tener su propia instancia de EntityManager y, al mismo tiempo, comparte una única EntityManagerFactory para toda la aplicación. Cuando la conexión a la base de datos ya no es necesaria, se puede cerrar el EntityManager:

```
em.cerrar();
```

Cerrar un EntityManager no cierra la base de datos en sí (ese es el trabajo de la fábrica como se explicó anteriormente).

Una vez cerrado el objeto EntityManager, no se puede reutilizar. Sin embargo, la instancia propietaria de EntityManagerFactory puede preservar los recursos de EntityManager (como un puntero de archivo de base de datos o un socket a un servidor remoto) en un grupo de conexiones y usarlos para acelerar la construcción futura de EntityManager.

Realizando una transacción

Las operaciones que modifican el contenido de la base de datos, como almacenar, actualizar y eliminar, solo deben realizarse dentro de una transacción activa. Dado un EntityManager, em, es muy fácil comenzar una transacción:

```
em.getTransaction().begin();
```

Existe una relación uno a uno entre una instancia de EntityManager y las instancias de EntityTransaction asociadas que devuelve el método getTransaction. Cuando una transacción está activa, puede invocar métodos de EntityManager que modifican el contenido de la base de datos, como persistir y eliminar. Las actualizaciones de la base de datos se recopilan y administran en la memoria y se aplican a la base de datos cuando se confirma la transacción:

```
em.getTransaction().commit();
```

4.2. Consultas a la base de objetos.

Hay varias formas de recuperar datos en ObjectDB, se puede utilizar JPA Query Structure (JPQL) o Criteria, por ejemplo:

Para recuperar un objeto de una clase a partir de la clave identificativa, por ejemplo, el ponente 1, utilizaremos:

```
Ponente ponente = em.find(Ponente.class, 1);
```

También podemos construir sentencias similares a SQL, utilizando JPQL. La interfaz TypedQuery, que se introdujo en JPA 2, es una *subinterfaz* de tipo seguro de Query y suele ser la forma preferida de trabajar con consultas. Por ejemplo, para recuperar todos los ponentes, usaríamos lo siguiente:

```
TypedQuery<Ponente> query =  
    em.createQuery("SELECT p FROM Ponente p", Ponente.class);  
List<Ponente> results = query.getResultList();
```

El método `getResultList` ejecuta la consulta y devuelve los objetos de resultado cuando se esperan múltiples resultados y la lista de resultados se puede iterar como cualquier colección Java normal. El método `getSingleResult` se utiliza cuando solo se obtendrá un resultado.

Se pueden usar consultas más avanzadas, como en SQL para recuperar objetos que cumplan filtros de la base de datos (usando una cláusula `WHERE`), ordenar los resultados (usando una cláusula `ORDER BY`) y agrupando los resultados (usando las cláusulas `GROUP BY` y `HAVING`). La sintaxis es:

```
SELECT ...  
FROM ...  
[WHERE ...]  
[GROUP BY ...][HAVING ...]  
[ORDER BY ...]
```

4.3. Actualización de objetos simples.

Un **objeto simple** es un objeto que no contiene a otros objetos, como por ejemplo los objetos de la clase **ponente**. Un segmento de la definición de esta clase, es la siguiente:

```
//clase que implementa la entidad ponente. Objeto Simple
public class ponente {
    private String nif;
    private String nombre;
    private String email;
    private float cache;
    //constructores
    + public ponente() {...}
    + public ponente(String ni, String n, String e) {...}
    + public ponente(String ni, String no, String e, float c) {...}
    //métodos básicos para asignar y obtener valores de atributos
    + public void setNif(String n) {...}
    + public String getNif() {...} _____
```

Para **consultar objetos simples** se pueden utilizar cualquiera de los sistemas de consulta proporcionados por ObjectDB, tal y como has podido ver en el apartado anterior.

Para **insertar** un objeto nuevo, basta con crearlo y persistirlo. En el siguiente ejemplo se guarda un nuevo ponente:

```
Ponente ponente = new Ponente("1", "Samuel", "samuel@correo.es", 100f);  
em.getTransaction().begin();  
em.persist(ponente);  
em.getTransaction().commit();
```

Para **eliminar** objetos almacenados utilizaremos el método **remove()** del EntityManager. Primero debe ser recuperado. Por ejemplo, el siguiente código elimina el objeto ponente que tiene el NIF 1.

```
Ponente ponente = em.find(Ponente.class, 1)  
em.getTransaction().begin();  
em.remove(ponente);  
em.getTransaction().commit();
```

ObjectDB necesita conocer previamente un objeto para poder actualizarlo. Esto significa que, para poder ser actualizados los objetos, éstos deben de haber sido insertados o recuperados en la misma sesión; en otro caso se añadirá otro objeto en vez de actualizarse.

El EntityManager que administra una entidad es responsable de detectar automáticamente los cambios (**actualización**) en el objeto de la entidad y aplicarlos a la base de datos cuando se confirma la transacción. Por ejemplo, para incrementar en 100 el caché de todos los ponentes que hay en la base de datos, se puede hacer lo siguiente:

```
TypedQuery<Ponente> query =  
    em.createQuery("SELECT p FROM Ponente p", Ponente.class);  
List<Ponente> results = query.getResultList();  
em.getTransaction().begin();  
for (Ponente p : results) {  
    p.setCache(p.getCache() + 100); //update entity  
}  
em.getTransaction().commit();
```

4.4. Actualización de objetos estructurados.

Los **objetos estructurados** son objetos que contienen a su vez a otros objetos (objetos hijo u objetos miembro).

En el caso de objetos estructurados se habla de diferentes **niveles de profundidad del objeto**. El nivel más alto, nivel 1, será el que corresponde a la definición del objeto estructurado (objeto padre), el siguiente nivel, nivel 2, corresponderá a la definición del objeto hijo y así sucesivamente podrá haber un nivel 3, 4... dependiendo de que los objetos hijos a su vez incluyan en su definición a otro u otros objetos miembro.

En el siguiente ejemplo, definimos la clase **charla** (objeto estructurado padre) que incorpora a un objeto **ponente** (objeto miembro). El nivel más alto de profundidad o nivel 1 es el que corresponde a la definición de charla y el nivel 2 corresponderá a la definición del objeto ponente.



¿Cómo se almacenan, consultan y actualizan los objetos estructurados en ObjectDB?

- Los objetos estructurados se almacenan asignando valores con **set()** y después persistiendo el objeto con **persist()**. Al almacenar un objeto estructurado del nivel más alto, se almacenarán de forma implícita todos los objetos hijo que tengan persistencia en cascada.
- Las consultas se realizan por cualquiera de los sistemas soportados por el gestor y se podrá ir descendiendo por los diferentes niveles de profundidad.
- La eliminación o borrado de un objeto estructurado se realiza mediante el método **remove()**. Por defecto, no se eliminarán los objetos miembro. Para eliminar objetos estructurados en cascada o de forma recursiva, eliminando los objetos miembro, habrá que configurar la persistencia en cascada.
- La modificación se realizará actualizando los nuevos valores mediante el método **set()**.

5. El lenguaje de consulta de objetos OQL.

OQL (Object Query Language) es el lenguaje de consulta de objetos propuesto en el estándar ODMG.

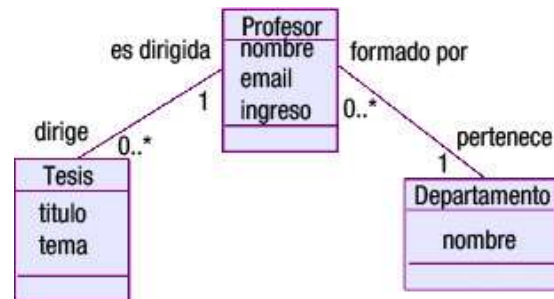
Las siguientes son algunas de las **características más relevantes de OQL**:

- Es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos.
- Su **sintaxis es similar a la de SQL**, proporcionando un superconjunto de la sintaxis de la sentencia **SELECT**, con algunas características añadidas para los conceptos ODMG, como la identidad del objeto, los objetos complejos, las operaciones, la herencia, el polimorfismo y las relaciones.
- **No posee primitivas para modificar el estado de los objetos** ya que las modificaciones se pueden realizar mediante los métodos que estos poseen.

- Puede ser usado como un **lenguaje autónomo o incrustado** dentro de otros lenguajes como C++, Smalltalk y Java.
- Una **consulta OQL incrustada** en uno de estos lenguajes de programación puede devolver objetos que coincidan con el sistema de tipos de ese lenguaje.
- Desde OQL se pueden invocar operaciones escritas en estos lenguajes.
- Permite **acceso tanto asociativo como *navegacional***:
 - Una **consulta asociativa** devuelve una colección de objetos.
 - Una **consulta *navegacional*** accede a objetos individuales y las interrelaciones entre objetos sirven para navegar entre objetos.

5.1. Sintaxis, expresiones y operadores.

La sintaxis básica y resumida de una sentencia **SELECT** del OQL estándar es la siguiente:



```
SELECT [DISTINCT] <expresión, ...>  
FROM <lista from>  
[WHERE <condición> ]  
[ORDER BY <expresión>]
```

Por ejemplo, suponiendo el esquema de base de objetos de la imagen anterior, la siguiente sentencia select recupera de la base de objetos los atributos nombre y el correo de objetos tipo profesor cuyo año de ingreso es anterior al 1990, y ordenados alfabéticamente por nombre:

```
SELECT p.nombre, p.email  
FROM p in Profesor  
WHERE p.ingreso <= 1990  
ORDER BY p.nombre;
```

Algunas **consideraciones a tener en cuenta:**

- En las consultas se necesita un punto de entrada, que suele ser el nombre de una clase.
- El resultado de una consulta es una colección que puede ser tipo **bag** (si hay valores repetidos) o tipo **set** (no hay valores repetidos). En este último caso habrá que especificar **SELECT DISTINCT**.

- En general, una consulta OQL puede devolver un resultado con una estructura compleja especificada en la misma consulta utilizando **struct**.
- Una vez que se establece un punto de entrada, se pueden utilizar expresiones de caminos para especificar un camino a atributos y objetos relacionados. Una expresión de camino empieza normalmente con un nombre de objeto persistente o una variable iterador, seguida de ninguno o varios nombres de relaciones o de atributos conectados mediante un punto.
- Es posible crear objetos mutables (no literales) formados por el resultado de una consulta.

Además, en una consulta OQL se pueden utilizar, entre otros, los siguientes **operadores y expresiones**:

- Operadores de acceso: **"."** / **"->"** aplicados a un atributo, una expresión o una relación.
- **FIRST / LAST** (primero / último elemento de una lista o un vector).
- Operadores aritméticos: **+**, **-**, *****, **/**, **-**(unario), **MOD**, **ABS** para formar expresiones aritméticas.
- Operadores relacionales: **>**, **<**, **>=**, **<=**, **<>**, **=** que permiten comparaciones y construir expresiones lógicas.
- Operadores lógicos: **NOT**, **AND**, **OR** que permiten enlazar otras expresiones.

A continuación, de manera resumida, vemos **algunas otras características** del estándar OQL:

- Definición de vistas, es decir, es posible dar nombre a una consulta y utilizarlo en otras consultas.
- Extracción de elementos sencillos de colecciones **set**, **bag**, o **list**.
- Operadores de colecciones como funciones de agregación (**MAX()**, **MIN()**, **COUNT()**, **SUM()** y **AVG()**) y cuantificadores (**FOR ALL**, **EXISTS**).
- Realización de agrupaciones mediante **GROUP BY** y filtro de los grupos mediante **HAVING**.
- Combinación de consultas mediante **JOINS**.
- Unión, intersección y resta de colecciones mediante los operadores **UNION**, **INTERSEC** y **EXCEPT**.

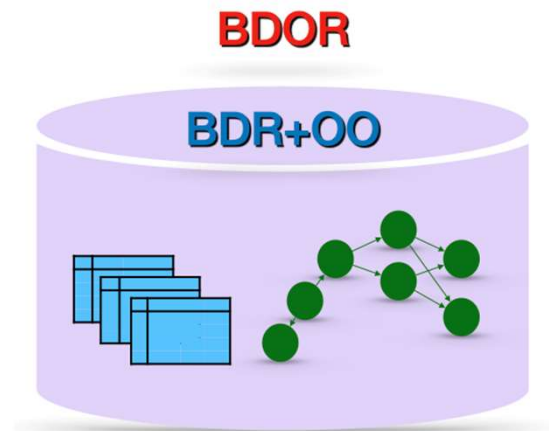
6. Características de las bases de datos objeto-relacionales.

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.

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.

7. Gestores de Bases de Datos Objeto-Relacionales.

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**:

- **Oracle.**
- **PostgreSQL.**
- **MySQL.**

Como base de datos objeto relacional a desarrollar se utilizará **ORACLE**.

El desarrollo del paradigma orientado a objetos aporta un gran cambio en el modo en que vemos los datos y los procedimientos que actúan sobre ellos. Tradicionalmente, los datos y los procedimientos se han almacenado separadamente: los datos y sus relaciones en la base de datos y los procedimientos en los programas de aplicación. La orientación a objetos, sin embargo, combina los procedimientos de una entidad con sus datos.

Esta combinación se considera como un paso adelante en la gestión de datos. Las entidades son unidades autocontenidas que se pueden reutilizar con relativa facilidad. En lugar de ligar el comportamiento de una entidad a un programa de aplicación, el comportamiento es parte de la entidad en si, por lo en cualquier lugar en el que se utilice la entidad, se comporta de un modo predecible y conocido.

El modelo orientado a objetos también soporta relaciones de muchos a muchos, siendo el primer modelo que lo permite. Aun así, se debe ser muy cuidadoso cuando se diseñan estas relaciones para evitar pérdidas de información.

Por otra parte, las **bases de datos orientadas a objetos son *navegacionales***: el acceso a los datos es a través de las relaciones, que se almacenan con los mismos datos. Esto se considera un paso atrás. Las bases de datos orientadas a objetos no son apropiadas para realizar consultas ad hoc, al contrario que las bases de datos relacionales, aunque normalmente las soportan. La naturaleza *navegacional* de las bases de datos orientadas a objetos implica que las consultas deben seguir relaciones predefinidas y que no pueden insertarse nuevas relaciones “al vuelo”.

No parece que las bases de datos orientadas a objetos vayan a reemplazar a las bases de datos relacionales en todas las aplicaciones del mismo modo en que estas reemplazaron a sus predecesoras.

Los objetos han entrado en el mundo de las bases de datos de formas:

- **SGBD orientados a objetos puros**: son SGBD basados completamente en el modelo orientado a objetos.
- **SGBD híbridos u objeto–relacionales**: son SGBD relacionales que permiten almacenar objetos en sus relaciones (tablas).

El modo en que los objetos han entrado en el mundo de las bases de datos relacionales es en forma de dominios, actuando como el tipo de datos de una columna. Los objetos y los literales se categorizan en **tipos**.

Cada tipo tiene un dominio específico compartido por todos los objetos y literales de ese tipo. Los **tipos** también pueden tener comportamientos. Cuando un tipo tiene comportamientos, todos los objetos de ese tipo comparten los mismos comportamientos.

Hay **dos implicaciones** muy importantes por el hecho de utilizar una **clase como un dominio**: Es posible almacenar múltiples valores en una columna de una misma fila ya que un objeto suele contener múltiples valores. Sin embargo, si se utiliza una clase como dominio de una columna, en cada fila esa columna solo puede contener un objeto de la clase (se sigue manteniendo la restricción del modelo relacional de contener valores atómicos en la intersección de cada fila con cada columna).

Es posible almacenar procedimientos en las relaciones porque un objeto está enlazado con el código de los procesos que sabe realizar (los métodos de su clase). Otro modo de incorporar objetos en las bases de datos relacionales es construyendo tablas de objetos, donde **cada fila es un objeto**. Ya que un sistema objeto–relacional es un sistema relacional que permite almacenar objetos en sus tablas, la base de datos sigue sujeta a las restricciones que se aplican a todas las bases de datos relacionales y conserva la capacidad de utilizar operaciones de concatenación (join) para implementar las relaciones “al vuelo”.

Un RDBMS (sistema gestor de bases de datos relacionales) que utiliza características orientadas a objetos, tales como tipos definidos por el usuario, herencia y polimorfismo, se conoce como ORDBMS (Sistema gestor de bases de datos relacionales con orientación a objetos).

7.1. Tipos de objetos.

El modelo relacional está diseñado para representar los datos como una serie de tablas con columnas y atributos. Oracle desde la versión 8 incorpora tecnologías orientadas a objetos. En este sentido, permite construir tipos de objetos complejos, entendidos como:

- Capacidad para definir objetos dentro de objetos.
- Cierta capacidad para encapsular o asociar métodos con dichos objetos.

Estructura de un tipo de objeto

Un **tipo de objeto** representa una **entidad del mundo real**. Encapsula datos y operaciones, por lo que en la especificación sólo se pueden declarar atributos y métodos, pero no constantes, excepciones, cursores o tipos. Al menos un atributo es requerido y los métodos son opcionales. Se compone de los siguientes elementos:

- Su **nombre** que sirve para identificar el tipo de los objetos.
- Sus **atributos** que modelan la estructura y los valores de los datos de ese tipo. Cada atributo puede ser de un tipo de datos básico o de un tipo de usuario.
- Sus **métodos** (procedimientos o funciones) escritos en lenguaje PL/SQL (almacenados en la BDOR), o escritos en C (almacenados externamente).

Los tipos de objetos actúan como plantillas para los objetos de cada tipo. A continuación, vemos un ejemplo de cómo definir el tipo de dato `Direccion_T` en el lenguaje de definición de datos de Oracle, y cómo utilizar este tipo de dato para definir el tipo de dato de los objetos de la clase de `Cliente_T`.


```
CREATE TYPE direccion_t AS OBJECT (  
  calle VARCHAR2(200),  
  ciudad VARCHAR2(200),  
  prov CHAR(2),  
  codpos VARCHAR2(20));  
CREATE TYPE cliente_t AS OBJECT (  
  clinum NUMBER,  
  clinomb VARCHAR2(200),  
  direccion direccion_t,  
  telefono VARCHAR2(20),  
  fecha_nac DATE,  
  MEMBER FUNCTION edad RETURN NUMBER,  
  PRAGMA  
  RESTRICT_REFERENCES(edad,WNDS));
```

Atributos

Como las variables, un **atributo** se declara mediante un nombre y un tipo. El **nombre debe ser único** dentro del tipo de objeto (aunque puede reutilizarse en otros objetos) y el tipo puede ser cualquier tipo de Oracle excepto:

- LONG y LONG RAW.
- NCHAR, NCLOB y NVARCHAR2.
- MLSLABEL y ROWID.
- Los tipos específicos de PL/SQL: BINARY_INTEGER (y cualquiera de sus subtipos), BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE y %ROWTYPE.
- Los tipos definidos en los paquetes PL/SQL.

Tampoco se puede inicializar un atributo en la declaración empleando el operador de asignación o la cláusula DEFAULT, del mismo modo que no se puede imponer la restricción NOT NULL. Sin embargo, los objetos se pueden almacenar en tablas de la base de datos en las que sí es posible imponer restricciones.

Métodos

En general, un método es un **subprograma** declarado en una especificación de tipo mediante la palabra clave MEMBER. El método no puede tener el mismo nombre que el tipo de objeto ni el de ninguno de sus atributos.

Muchos métodos constan de **dos partes**: especificación y cuerpo. La **especificación** consiste en el nombre del método, una lista opcional de parámetros y en el caso de funciones un tipo de retorno. El **cuerpo** es el código que se ejecuta para llevar a cabo una operación específica.

Para cada especificación de método en una especificación de tipo debe existir el correspondiente cuerpo del método. El PL/SQL compara la especificación del método y el cuerpo token a token, por lo que las cabeceras deben coincidir palabra a palabra.

En un tipo de objeto, los métodos pueden hacer referencia a los atributos y a los otros métodos sin cualificador. Los métodos se pueden ejecutar sobre los objetos de su mismo tipo. Si x es una variable PL/SQL que almacena objetos del tipo Cliente_T, entonces x.edad() calcula la edad del cliente almacenado en x. La definición del cuerpo de un método en PL/SQL se hace de la siguiente manera:

```
CREATE OR REPLACE TYPE BODY cliente_t AS
MEMBER FUNCTION edad RETURN NUMBER IS
    a NUMBER;
    d DATE;
BEGIN
    d:= today();
    a:= d.año - fecha_nac.año;
    IF (d.mes < fecha_nac.mes) OR
    ((d.mes = fecha_nac.mes) AND (d.dia < fecha_nac.dia))
    THEN a:= a-1;
    END IF;
    RETURN a;
END;
END;
```

El parámetro SELF

Todos los métodos de un tipo de objeto aceptan como primer parámetro una instancia predefinida del mismo tipo denominada **SELF**. Independientemente de que se declare implícita o explícitamente, SELF es siempre el primer parámetro pasado a un método. Por ejemplo, el método **transform** declara SELF como un parámetro IN OUT:

```
CREATE TYPE Complex AS OBJECT (  
  MEMBER FUNCTION transform ( SELF IN OUT Complex ) . . .
```

El modo de acceso por omisión de SELF, es decir, cuando no se declara explícitamente es:

- En **funciones** miembro el acceso de SELF es **IN**.
- En **procedimientos**, si SELF no se declara, su modo por omisión es **IN OUT**.

En el cuerpo de un método, SELF denota al objeto a partir del cual se invocó el método. Como muestra el siguiente ejemplo, los métodos pueden hacer referencia a los atributos de SELF sin necesidad de utilizar un cualificador:

```

CREATE FUNCTION gcd ( x INTEGER , y INTEGER ) RETURN INTEGER AS
-- Encuentra el maximo comun divisor de x e y
ans INTEGER;
BEGIN
IF x < y THEN ans := gcd ( x , y ) ;
ELSE ans := gcd ( y , x MOD y ) ;
ENDIF ;
RETURN ans ;
END;
/

CREATE TYPE Rational AS
num INTEGER ,
den INTEGER ,
MEMBER PROCEDURE normalize ,
. . .
) ;
/

CREATE TYPE BODY Rational AS
MEMBER PROCEDURE normalize IS
    g INTEGER;
BEGIN
    --Estas dos sentencias son equivalentes
    g := gcd ( SELF . num , SELF . den ) ;
    g := gcd ( num , den ) ;
    num := num / g ;
    den := den / g ;
END normalize ;
. . .
END;
/

```

Constructores

En Oracle, todos los tipos de objetos tienen asociado por defecto un método que construye nuevos objetos de ese tipo de acuerdo a la especificación del tipo. El nombre del método coincide con el nombre del tipo, y sus parámetros son los atributos del tipo. Por ejemplo, las siguientes expresiones construyen dos objetos con todos sus valores.

```
direccion_t('Avenida Sagunto', 'Puzol', 'Valencia', 'E-23523')  
cliente_t( 2347, 'Juan Pérez Ruíz', direccion_t('Calle Eo', 'Onda', 'Castellón', '34568'), '696-  
779789', '12/12/1981')
```

Métodos de comparación

Para comparar los objetos de cierto tipo es necesario indicar a Oracle cuál es el criterio de comparación. Para ello, hay que escoger entre un método MAP u ORDER, debiéndose definir al menos uno de estos métodos por cada tipo de objeto que necesite ser comparado. La diferencia entre ambos es la siguiente:

- Un **método MAP** sirve para indicar cuál de los atributos del tipo se utilizará para ordenar los objetos del tipo, y por tanto se puede utilizar para comparar los objetos de ese tipo por medio de los operadores de comparación aritméticos (<, >).

El PL/SQL usa esta función para evaluar expresiones booleanas como $x > y$ y para las comparaciones implícitas que requieren las cláusulas DISTINCT, GROUP BY y ORDER BY. Un tipo de objeto puede contener sólo una función de MAP, que necesariamente debe carecer de parámetros y debe devolver uno de los siguientes tipos escalares: DATE, NUMBER, VARCHAR2 y cualquiera de los tipos ANSI SQL (como CHARACTER o REAL). Por ejemplo, la siguiente declaración permite decir que los objetos del tipo cliente_t se van a comparar por su atributo clinum.

```
CREATE OR REPLACE TYPE BODY cliente_t AS
MAP MEMBER FUNCTION ret_value RETURN NUMBER IS
BEGIN
    RETURN clinum;
END;
END;
```


- 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 devolverá un valor **negativo** si el parámetro de entrada es **mayor que el atributo**, un valor **positivo** si ocurre lo **contrario** y un **cero** si ambos son **iguales**. El siguiente ejemplo define un orden para el tipo cliente_t diferente al anterior. Sólo una de estas definiciones puede ser válida en un tiempo dado.

```
CREATE TYPE cliente_t AS OBJECT (  
    clinum NUMBER,  
    clinomb VARCHAR2(200),  
    direccion direccion_t,  
    telefono VARCHAR2(20),  
    fecha_nac DATE,  
    ORDER MEMBER FUNCTION  
    cli_ordenados (x IN clientes_t) RETURN INTEGER,  
  
CREATE OR REPLACE TYPE BODY cliente_t AS  
    ORDER MEMBER FUNCTION cli_ordenados (x IN cliente_t) RETURN INTEGER IS  
BEGIN  
    RETURN clinum - x.clinum; /*la resta de los dos números clinum*/  
END;  
END;
```

Veamos otro ejemplo: supongamos que `c1` y `c2` son objetos del tipo `Customer`. Una comparación del tipo `c1 > c2` invoca al **método `match`** automáticamente. El método devuelve un número negativo, cero o positivo que indica que `SELF` es respectivamente menor, igual o mayor que el otro parámetro:

```
CREATE TYPE Customer AS OBJECT (  
    id NUMBER,  
    name VARCHAR2( 20 ),  
    addr VARCHAR2( 30 ),  
    ORDER MEMBER FUNCTION match ( c Customer ) RETURN INTEGER  
);  
/  
CREATE TYPE BODY Customer AS  
    ORDER MEMBER FUNCTION match ( c Customer ) RETURN INTEGER IS  
BEGIN  
    IF id < c.id THEN  
        RETURN -1;  
        --Cualquier numero negativo .  
    ELSEIF id > c.id THEN  
        RETURN 1;  
        --cualquier numero positivo  
    ELSE  
        RETURN 0;  
    END IF;  
END;  
END;  
/
```

Un tipo de objeto puede contener un **único método ORDER**, que es una función que devuelve un resultado numérico. Es importante tener en cuenta los siguientes puntos:

- Un método MAP proyecta el valor de los objetos en valores escalares (que son más fáciles de comparar). Un método ORDER simplemente compara el valor de un objeto con otro.
- Se puede declarar un método MAP o un método ORDER, pero no ambos.
- Si se declara uno de los dos métodos, es posible comparar objetos en SQL o en un procedimiento.
- Cuando es necesario ordenar un número grande de objetos es mejor utilizar un método MAP (ya que una llamada por objeto proporciona una proyección escalar que es más fácil de ordenar). Un método ORDER es menos eficiente: debe invocarse repetidamente ya que compara sólo dos objetos cada vez.

Si un tipo de objeto no tiene definido ninguno de estos métodos, Oracle es incapaz de deducir cuándo un objeto es mayor o menor que otro.

Sin embargo, sí puede determinar cuándo dos objetos del mismo tipo son iguales. Para ello, el sistema compara el valor de los atributos de los objetos uno a uno:

- Si todos los atributos son no nulos e iguales, Oracle indica que ambos objetos son iguales.
- Si alguno de los atributos no nulos es distinto en los dos objetos, entonces Oracle dice que son diferentes.
- En otro caso, Oracle dice que no puede comparar ambos objetos.

Declaración e inicialización de objetos

Una vez que se ha definido un tipo de objeto y se ha instalado en el esquema de la base de datos, es posible usarlo en cualquier bloque PL/SQL. Las instancias de los objetos se crean en tiempo de ejecución. Estos objetos siguen las reglas normales de ámbito y de instanciación. En un bloque o subprograma, los objetos locales son instanciados cuando se entra en el bloque o *subprograma* y dejan de existir cuando se sale. En un paquete, los objetos se instancian cuando se referencia por primera vez al paquete y dejan de existir cuando finaliza la sesión.

Declaración de objetos

Los tipos de objetos se declaran del mismo modo que cualquier tipo interno. Por ejemplo, en el bloque que sigue se declara un objeto *r* de tipo Racional y se invoca al constructor para asignar su valor.

La llamada asigna los valores 6 y 8 a los atributos *num* y *den* respectivamente:

```
DECLARE
r
Racional ;
BEGIN
r
:= Racional ( 6 , 8 ) ;
DBMS_OUTPUT.
PUT_LINE( r . num) ;
```

También es posible declarar objetos como **parámetros formales de funciones y procedimientos**, de modo que es posible pasar objetos a los subprogramas almacenados y de un subprograma a otro. En el siguiente ejemplo, se emplea un objeto de tipo Account para especificar el tipo de dato de un parámetro formal:

```
DECLARE
. . .
PROCEDURE open_act ( new_acct IN OUT Account ) IS . . .
```

y en el siguiente ejemplo se declara una función que devuelve un objeto de tipo Account:

```
DECLARE
. . .
FUNCTION get_act   ( act_id IN INTEGER ) RETURN Account IS . . .
```

Inicialización de objetos

Hasta que se inicializa un objeto, invocando al constructor para ese tipo de objeto, el objeto se dice que es **atómicamente nulo**. Esto es, el objeto es nulo, no sólo sus atributos.

Un objeto nulo siempre es diferente a cualquier otro objeto. De hecho, la comparación de un objeto nulo con otro objeto siempre resulta NULL. Del mismo modo, si se asigna un objeto con otro objeto atómicamente nulo, el primero se convierte a su vez en un objeto atómicamente nulo (y para poder utilizarlo debe ser reinicializado).

En resumen, si asignamos el no-valor NULL a un objeto, éste se convierte en atómicamente nulo, como se ilustra en el siguiente ejemplo:

```
DECLARE
r Racional ;
BEGIN
r Racional := Racional ( 1 , 2 ) ; --r = 1 / 2
r := NULL;--r atomicament e nulo
IF r IS NULL THEN . . .-- la condición resulta a TRUE
```


Una buena práctica de programación consiste en inicializar los objetos en su declaración, como se muestra en el siguiente ejemplo:

```
DECLARE  
r Racional := Racional ( 2 , 3 ) ; --r = 2 / 3
```

Objetos sin inicializar en PL/SQL

PL/SQL se comporta del siguiente modo cuando accede a objetos sin inicializar:

- Los atributos de un objeto no inicializado se evalúan en cualquier expresión como NULL.
- Intentar asignar valores a los atributos de un objeto sin inicializar provoca la excepción predefinida `ACCESS_INTO_NULL`.
- La operación de comparación `IS NULL` siempre produce `TRUE` cuando se aplica a un objeto no inicializado o a cualquiera de sus atributos.

Existe, por tanto, una sutil diferencia entre **objetos nulos** y **objetos con atributos nulos**. El siguiente ejemplo intenta ilustrar esa diferencia:

```
DECLARE
r Racional ;--r es atómicamente nulo
BEGIN
IF r IS NULL THEN . . . --TRUE
IF r . num IS NULL THEN . . . --TRUE
r := Racional (NULL , NULL) ; --Inicializar
r . num = 4 ;--Ex i t o : r ya no es atómicamente nulo aunque
--sus atributos son nulos
r := NULL;--r es de nuevo atómicamente nulo
r . num := 4 ;--Provoca la excepción ACCESS_INTO_NULL
EXCEPTION
WHEN ACCESS_INTO_NULL THEN
. . .
END;
/
```

La invocación de los métodos de un objeto no inicializado está permitida, pero en este caso: SELF toma el valor NULL. Cuando los atributos de un objeto no inicializado se pasan como parámetros IN, se evalúan como NULL. Cuando los atributos de un objeto no inicializado se pasan como parámetros OUT o IN OUT, se produce una excepción si se intenta asignarles un valor.

Acceso a los atributos

Para acceder o cambiar los valores de un atributo se emplea la notación punto ('.'). El siguiente ejemplo ilustra esa notación:

```
DECLARE
r RAcional := RAcional (NULL , NULL) ;
    numerador INTEGER;
    denominador INTEGER;
BEGIN
    . . .
    denominador := r . den ;
    r . num = numerador ;
```

Los nombres de los atributos pueden encadenarse, lo que permite acceder a los atributos de un tipo de objeto anidado. Por ejemplo, supongamos que definimos los tipos de objeto Address y Student como sigue:

```

CREATE TYPE Address AS OBJECT (
  street VARCHAR2( 30 ) ,
  city VARCHAR2( 20 ) ,
  state CHAR( 2 ) ,
  zip_code VARCHAR2( 5 )
) ;
/
CREATE TYPE Student AS OBJECT (
  name VARCHAR2( 20 ) ,
  home_address Address ,
  phone_number VARCHAR2( 10 ) ,
  status VARCHAR2( 10 ) ,
  advisor_name VARCHAR2( 20 ) ,
  . . .
) ;
/

```

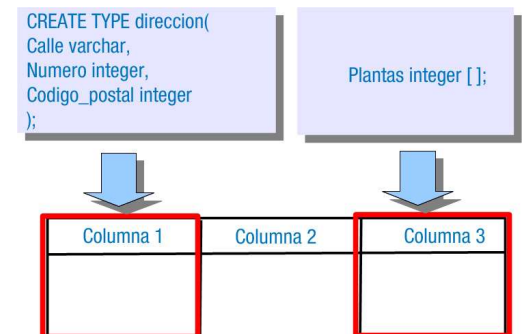
Observa que zip_code es un atributo de tipo Address y que Address es el tipo de dato del atributo home_address del tipo de objeto Student. Si s es un objeto Student, para acceder al valor de su zip_code se emplea la siguiente notación:

```
s . home_address . zip_code
```

7.2. Tablas de objetos

Una vez definidos los tipos, éstos pueden utilizarse para definir nuevos tipos, tablas que almacenen objetos de esos tipos, o para definir el tipo de los atributos de una tabla. Una **tabla de objetos** es una clase especial de tabla que almacena un objeto en cada fila y que facilita el acceso a los atributos de esos objetos como si fueran columnas de la tabla. Por ejemplo, se puede definir una tabla para almacenar los clientes de este año y otra para almacenar los de años anteriores de la siguiente manera:

```
CREATE TABLE clientes_año_tab OF cliente_t
(clinum PRIMARY KEY);
CREATE TABLE clientes_antiguos_tab (
año NUMBER,
cliente cliente_t);
```



La diferencia entre la primera y la segunda tabla es que la primera almacena objetos con su propia identidad (OID) y la segunda no es una tabla de objetos, sino una tabla con una columna con un tipo de datos de objeto. Es decir, la segunda tabla tiene una columna con un tipo de datos complejo, pero sin identidad de objeto. Además de esto, **Oracle** permite considerar una **tabla de objetos** desde dos puntos de vista:

- Como una tabla con una sola columna cuyo tipo es el de un tipo de objetos.
- Como una tabla que tiene tantas columnas como atributos los objetos que almacena.

Por ejemplo, se puede ejecutar una de las dos instrucciones siguientes. En la primera instrucción, la tabla `clientes_año_tab` se considera como una tabla con varias columnas cuyos valores son los especificados. En el segundo caso, se la considera como con una tabla de objetos que en cada fila almacena un objeto. En esta instrucción la cláusula `VALUE` permite visualizar el valor de un objeto.

```
INSERT INTO clientes_año_tab VALUES( 2347, 'Juan Pérez Ruíz', direccion_t ('Calle Castalia', 'Onda', 'Castellón', '34568'), '696-779789', '12/12/1981');

SELECT VALUE(c)
FROM clientes_año_tab c
WHERE c.clinomb = 'Juan Pérez Ruíz';
```

Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, sólo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

7.3. Referencias entre objetos

Los **identificadores únicos** asignados por Oracle a los objetos que se almacenan en una tabla, permiten que éstos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. El tipo de datos proporcionado por Oracle para soportar esta facilidad se denomina **REF**. Un atributo de tipo REF almacena una referencia a un objeto del tipo definido, e implementa una relación de asociación entre los dos tipos de objetos. Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos; sin embargo, no es posible operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre REF o NULL.

Cuando se define una columna de un tipo a REF, es posible restringir su dominio a los objetos que se almacenen en cierta tabla. Si la referencia no se asocia a una tabla, sino que sólo se restringe a un tipo de objeto, se podrá actualizar a una referencia a un objeto del tipo adecuado con independencia de la tabla donde se almacene. En este caso su almacenamiento requerirá más espacio y su acceso será menos eficiente. El siguiente ejemplo define un atributo de tipo REF y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF cliente_t;  
CREATE TYPE ordenes_t AS OBJECT (  
    ordnum NUMBER,  
    cliente REF clientes_t,  
    fechpedido DATE,  
    direcentrega direccion_t);  
CREATE TABLE ordenes_tab OF ordenes_t (  
    PRIMARY KEY (ordnum));
```

No es posible navegar a través de referencias en procedimientos SQL. Para esto es necesario utilizar el operador **DEREF** (abreviatura del término inglés dereference : *desreferenciar* un puntero es obtener el valor al cual apunta). Deref toma como argumento una referencia a un objeto y devuelve el valor de dicho objeto. Si la referencia está colgada, Deref devuelve el valor NULL.

En el ejemplo que sigue se *desreferencia* una referencia a un objeto Person de la tabla DUAL2. En estas circunstancias, no es necesario especificar una tabla de objetos ni un criterio de búsqueda ya que cada objeto almacenado en una tabla de objetos cuenta con un identificador de objeto único e inmutable que es parte de cada referencia a un objeto.

```
DECLARE
p1    Person ;
p_ref REF Person ;
name VARCHAR2(15) ;
BEGIN

...

/*Supongamos que p_ref  contiene una referencia valida a un objeto almacenado en una tabla de objetos*/

SELECT Deref( p_ref) INTO p1 FROM DUAL;
name := p1.last name ;
```

Es posible utilizar el operador DREF en sentencias SQL sucesivas para *desreferencias* como se muestra en el siguiente ejemplo:

```
CREATE TYPE PersonRef AS OBJECT (pref REF Person) ;
/
DECLARE
name VARCHAR2(15) ;
pr_ref REF PersonRef ;
pr PersonRef ;
p Person ;
BEGIN
. . .
/*Supongamos que pr_ref contiene una referencia valida*/
SELECT Deref( pr_ref ) INTO pr FROM DUAL;
SELECT Deref( pr.p_ref ) INTO p FROM DUAL;
name := p.last_name ;
...
END;
/
```

7.4. Tipos de datos colección

Para poder implementar **relaciones 1:N**, Oracle permite definir **tipos colección**. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera, es posible almacenar en un atributo un conjunto de tuplas en forma de array (**VARRAY**), o en forma de tabla anidada.

Al igual que los tipo objeto, los tipo colección también tienen por defecto unas funciones constructoras de colecciones cuyo nombre coincide con el del tipo. Los argumentos de entrada de estas funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor del tipo colección.

En Oracle es posible diferenciar entre un valor nulo y una colección vacía. Para construir una colección sin elementos se puede utilizar la función constructora del tipo seguida por dos paréntesis sin elementos dentro.

El tipo VARRAY

Un array es un conjunto ordenado de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del array. Oracle permite que los **VARRAY** sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando se declara el tipo VARRAY. Las siguientes declaraciones crean un tipo para una lista ordenada de precios, y un valor para dicho tipo.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12);  
precios('35', '342', '3970');
```

Se puede utilizar el tipo VARRAY para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objeto.
- Para definir una variable PL/SQL, un parámetro o el tipo que devuelve una función.

Cuando se declara un tipo VARRAY no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4000 bytes) se almacena aparte de la tabla como un BLOB.

En el siguiente ejemplo, se define un tipo de datos para almacenar una lista ordenada de teléfonos: el tipo list (ya que en el tipo set no existe orden). Este tipo se utiliza después para asignárselo a un atributo del tipo de objeto cliente_t.

```
CREATE TYPE lista_tel_t AS VARRAY(10) OF VARCHAR2(20) ;  
CREATE TYPE cliente_t AS OBJECT (clinum NUMBER, clinomb VARCHAR2(200),  
                                direccion direccion_t, lista_tel lista_tel_t );
```

La principal limitación del tipo VARRAY es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un VARRAY solamente pueden ser accedidos y recuperados como un bloque. Es decir, no se puede acceder individualmente a los elementos de un VARRAY. Sin embargo, desde un programa PL/SQL si que es posible definir un bucle que itere sobre los elementos de un VARRAY. Se define el tipo objeto ordenes_t y su atributo pedido almacena una tabla anidada del tipo lineas_pedido_t.

Tablas anidadas

Una **tabla anidada** es un conjunto de elementos del mismo tipo sin ningún orden predefinido. Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objeto definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo de objeto.

El siguiente ejemplo declara una tabla que después será anidada en el tipo `ordenes_t`. Los pasos de todo el diseño son los siguientes:

1. Se define el tipo de objeto `linea_t` para las filas de la tabla anidada.

```
CREATE TYPE linea_t AS OBJECT (  
  linum NUMBER,  
  item VARCHAR2(30),  
  cantidad NUMBER,  
  descuento NUMBER(6,2));
```

2. Se define el tipo colección tabla `lineas_pedido_t` para después anidarla.

```
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
```

Esta definición permite utilizar el tipo colección `lineas_pedido_t` para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objetos.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

3. Se define el tipo objeto `ordenes_t` y su atributo `pedido` almacena una tabla anidada del tipo `lineas_pedido_t`.

```
CREATE TYPE ordenes_t AS OBJECT (  
  ordnum NUMBER,  
  cliente REF cliente_t,  
  fechpedido DATE,  
  fechentrega DATE,  
  pedido lineas_pedido_t,  
  direcentrega direccion_t) ;
```


4. Se define la tabla de objetos ordenes_tab y se especifica la tabla anidada del tipo lineas_pedido_t.

```
CREATE TABLE ordenes_tab OF ordenes_t  
(ordnum PRIMARY KEY,  
SCOPE FOR (cliente) IS clientes_tab)  
NESTED TABLE pedido STORE AS pedidos_tab) ;
```

Este último paso es necesario realizarlo porque la declaración de una tabla anidada no reserva ningún espacio para su almacenamiento. Lo que se hace es indicar en qué tabla (pedidos_tab) se deben almacenar todas las líneas de pedido que se representen en el atributo pedido de cualquier objeto de la tabla ordenes_tab. Es decir, todas las líneas de pedido de todas las ordenes se almacenan externamente a la tabla de ordenes, en otra tabla especial. Para relacionar las tuplas de una tabla anidada con la tupla a la que pertenecen, se utiliza una columna oculta que aparece en la tabla anidada por defecto. Todas las tuplas de una tabla anidada que pertenecen a la misma tupla tienen el mismo valor en esta columna (NESTED_TABLE_ID).

A diferencia de los VARRAY, los elementos de las tablas anidadas (NESTED_TABLE) sí pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos.

Mas adelante, veremos, una forma conveniente de acceder individualmente a los elementos de una tabla anidada mediante un cursor anidado. Además, las tablas anidadas pueden estar indexadas.

7.5. Acceso a datos

Algunas cuestiones relacionadas con el acceso a datos:

Alias

En una base de datos con tipos y objetos, lo más recomendable es utilizar siempre alias para los nombres de las tablas. El alias de una tabla debe ser único en el contexto de la consulta. Los alias sirven para acceder al contenido de la tabla, pero hay que saber utilizarlos adecuadamente en las tablas que almacenan objetos.

El siguiente ejemplo ilustra cómo se deben utilizar.

```
CREATE TYPE persona AS OBJECT (nombre VARCHAR(20));  
CREATE TABLE ptab1 OF persona;  
CREATE TABLE ptab2 (c1 persona);  
CREATE TABLE ptab3 (c1 REF persona);
```

La diferencia entre las dos primeras tablas está en que la primera almacena objetos del tipo persona, mientras que la segunda tabla tiene una columna donde se almacenan valores del tipo persona. Considerando ahora las siguientes consultas, se ve cómo se puede acceder a estas tablas.

1. `SELECT nombre FROM ptab1;` Correcto
2. `SELECT c1.nombre FROM ptab2;` Incorrecto
3. `SELECT p.c1.nombre FROM ptab2 p;` Correcto
4. `SELECT p.c1.nombre FROM ptab3 p;` Correcto
5. `SELECT p.nombre FROM ptab3 p;` Incorrecto

En la primera consulta nombre es considerado como una de las columnas de la tabla ptab1, ya que los atributos de los objetos se consideran columnas de la tabla de objetos. Sin embargo, en la segunda consulta se requiere la utilización de un alias para indicar que nombre es el nombre de un atributo del objeto de tipo persona que se almacena en la columna c1. Para resolver este problema no es posible utilizar los nombres de las tablas directamente: ptab2.c1.nombre es incorrecto. Las consultas 4 y 5 muestran cómo acceder a los atributos de los objetos referenciados desde un atributo de la tabla ptab3.

En conclusión, para facilitar la formulación de consultas y evitar errores se recomienda utilizar alias para acceder a todas las tablas que contengan objetos con o sin identidad, y para acceder a las columnas de las tablas en general.

Inserción de referencias

La inserción de objetos con referencias implica la utilización del operador REF para poder insertar la referencia en el atributo adecuado. La siguiente sentencia inserta una orden de pedido en la tabla definida en la sección 1.1.3.

```
INSERT INTO ordenes_tab
SELECT 3001, REF(C), '30-MAY-1999', NULL
--se seleccionan los valores de los 4 atributos de la tabla
FROM cliente_tab C WHERE C.clinum= 3;
```

El acceso a un objeto desde una referencia REF requiere primero referenciar al objeto. Para realizar esta operación, Oracle proporciona el operador Deref. No obstante, utilizando la notación de punto también se consigue referenciar a un objeto de forma implícita. Observemos el siguiente ejemplo:

```
CREATE TYPE persona_t AS OBJECT (
nombre VARCHAR2(30),
jefe REF persona_t ) ;
```

Si x es una variable que representa a un objeto de tipo persona_t, entonces las dos expresiones siguientes son equivalentes:

1. x.jefe.nombre
2. y.nombre, y=DEREF(x.jefe)

Para obtener una referencia a un objeto de una tabla de objetos, se puede aplicar el operador REF como muestra el siguiente ejemplo:

```
CREATE TABLE persona_tab OF persona_t;  
DECLARE ref_persona REF persona_t;  
SELECT REF(pe) INTO ref_persona  
FROM persona_tab pe WHERE pe.nombre= 'Juan Pérez Ruíz';
```

Simétricamente, para recuperar un objeto desde una referencia es necesario usar Deref, como muestra el siguiente ejemplo que visualiza los datos del jefe de la persona indicada:

```
SELECT Deref(pe.jefe)  
FROM persona_tab pe WHERE pe.nombre= 'Juan Pérez Ruíz';
```

Llamadas a métodos

Para invocar un método hay que utilizar su nombre y unos paréntesis que encierren sus argumentos de entrada. Si el método no tiene argumentos, se especifican los paréntesis aunque estén vacíos. Por ejemplo, si `tb` es una tabla con la columna `c` de tipo de objeto `t`, y `t` tiene un método `m` sin argumentos de entrada, la siguiente consulta es correcta:

```
SELECT p.c.m FROM tb p;
```

Inserción en tablas anidadas

Además del constructor del tipo de colección disponible por defecto, la inserción de elementos dentro de una tabla anidada puede hacerse siguiendo estas dos etapas:

1. Crear el objeto con la tabla anidada y dejar vacío el campo que contiene las tuplas anidadas.
2. Comenzar a insertar tuplas en la columna correspondiente de la tupla seleccionada por una *subconsulta*.

Para ello, se tiene que utilizar la palabra clave THE con la siguiente sintaxis:

```
INSERT INTO THE (subconsulta) (tuplas a insertar)
```


Esta técnica es especialmente útil si dentro de una tabla anidada se guardan referencias a otros objetos. El siguiente ejemplo ilustra la manera de realizar estas operaciones sobre la tabla de ordenes (ordenes_tab) definida anteriormente.

```
INSERT INTO ordenes_tab --inserta una orden
SELECT 3001, REF(C), SYSDATE, '30-MAY-1999', lineas_pedido_t(), NULL
FROM cliente_tab C
WHERE C.clinum= 3 ;

INSERT INTO THE ( --selecciona el atributo pedido de la orden
SELECT P.pedido
FROM ordenes_tab P
WHERE P.ordnum = 3001
)
VALUES (linea_t(30, NULL, 18, 30));
--inserta una línea de pedido anidada
```

Para poner condiciones a las tuplas de una tabla anidada, se pueden utilizar cursores dentro de un SELECT o desde un programa PL/SQL. Veamos aquí un ejemplo de acceso con cursores. Utilizando un ejemplo anterior, vamos a recuperar el número de las ordenes, sus fechas de pedido y las líneas de pedido que se refieran al ítem 'CH4P3'.

```
SELECT ord.ordnum, ord.fechpedido,  
CURSOR (SELECT * FROM TABLE(ord.pedido) lp WHERE lp.item= 'CH4P3')  
FROM ordenes_tab ord;
```

La cláusula THE también sirve para seleccionar las tuplas de una tabla anidada. La sintaxis es como sigue:

```
SELECT ... FROM THE (subconsulta) WHERE ...
```

Por ejemplo, para seleccionar las primeras dos líneas de pedido de la orden 8778 se hace:

```
SELECT lp FROM THE  
(SELECT ord.pedido FROM ordenes_tab ord WHERE ord.ordnum= 8778) lp  
WHERE lp.linum<3;
```

8. Gestión de transacciones en las BDOR.

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 disminuye 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).

Anexo I. Estándar ODMG-93 u ODMG.

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.

Fue 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 elementos que permite el acceso por posición.

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