

MP0486

**Acceso a datos
UF6. Bases de datos
orientadas a objetos**

6.4. Lenguaje de consultas para objetos

Índice

☰	Objetivos	3
☰	Introducción	4
☰	Ejemplos de consultas OQL	5
☰	Preparación del proyecto en Eclipse	7
☰	Consultas con la librería NeoDatis	9
☰	Ejemplos resueltos	12
☰	Ordenar los resultados	13
☰	La clase And	14
☰	La clase Or	15
☰	Combinar And y Or	17
☰	Resumen	19

Objetivos

En esta lección perseguimos los siguientes objetivos:

1

Comprender el significado del estándar OQL para las consultas en bases de datos orientadas a objetos.

2

Desarrollar programas Java que ejecuten consultas sobre bases de datos orientadas a objetos.

¡Ánimo y adelante!

Introducción

OQL (*Object Query Language*) es un estándar para creación de consultas en bases de datos orientadas a objetos que nace a partir de SQL (*Structured Query Language*).

Como estándar, fue desarrollado por la organización Object Data Management Group.

OQL todavía está en vías de desarrollo, ya que debido a su complejidad ningún creador de software lo ha implementado completamente.

- i No hay que olvidar que se trata de un estándar, de modo que cada fabricante desarrolla OQL a su manera cumpliendo las normas que dicta el estándar.

La idea consiste en aprovechar la sintaxis de cada una de las sentencias del lenguaje SQL estándar para crear una sintaxis lo más parecida posible, pero trasladada al trabajo con objetos.

Ejemplos de consultas OQL

Para que comprendas mejor la sintaxis de las consultas OQL, vamos a ver algunos **ejemplos comparados con el estándar SQL**.

1

Ejemplo 1

- **SQL estándar:**

```
SELECT * FROM Producto  
WHERE nombre="Arenque ahumado";
```

- **OQL:**

```
select p from p in Producto  
where p.nombre="Arenque ahumado";
```

La variable *p* contendrá la referencia de cada objeto leído de la clase *Producto*. En el ejemplo, recuperaremos un objeto de la clase *Producto* completo, en concreto, un producto cuyo nombre sea "Arenque ahumado".

2

Ejemplo 2

- **SQL estándar:**

```
SELECT precio FROM Producto  
WHERE nombre="Arenque ahumado";
```

- **OQL:**

```
select p.precio from p in Producto  
where p.nombre="Arenque ahumado";
```

Este ejemplo es similar al anterior, pero en lugar de recuperar el objeto *Producto* completo, sólo recuperamos un dato elemental, el valor del atributo *precio*.

3

Ejemplo 3

- **SQL estándar:**

```
SELECT nombre, precio, stock
FROM Producto
WHERE nombre = "Arenque ahumado";
```

- **OQL:**

```
select struct(nom: p.nombre, pre: p.precio, st: p.stock)
from p in Producto
where p.nombre = "Arenque ahumado";
```

La variable *p* sigue conteniendo la referencia al objeto *Producto*, pero ahora no deseamos recuperar el objeto *Producto* completo, sino un subconjunto (estructura) formado por los atributos *nombre*, *precio* y *stock*. En las variables *nom*, *pre* y *st* se guardarán los valores de los atributos *nombre*, *precio* y *stock*.

4

Ejemplo 4

- **SQL estándar:**

```
SELECT * FROM Producto
WHERE precio > 50
ORDER BY precio;
```

- **OQL:**

```
select p from p in Producto
where p.precio > 50
order by p.precio;
```

En este caso, recuperaremos los objetos de la clase *Producto* cuyo *precio* sea mayor a 50€.

Preparación del proyecto Eclipse

Para que puedas poner en práctica los ejemplos que te mostraremos en los próximos apartados, primero tendrás que preparar el proyecto estándar java en Eclipse con los recursos necesarios.

¡Vamos manos a la obra!

1

Crea un proyecto estándar Java con el nombre que deseas (*File / New / Java Project*).

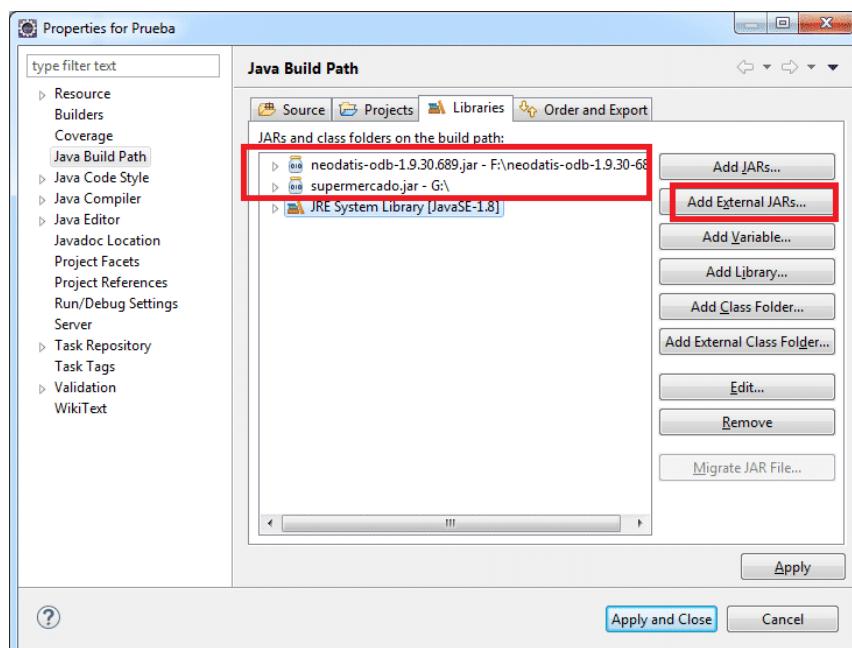
2

Trabajarás con la base de datos *ALMACEN.DB* que importaste en la lección anterior, pero necesitarás disponer de las clases *Categoría* y *Producto* para poder recoger los resultados de las consultas OQL que ejecutes. Para facilitarte la labor, hemos empaquetado dichas clases en una librería denominada *supermercado.jar* que podrás descargar en el siguiente enlace. Una vez descargado, renombra el archivo para que mantenga el nombre de *supermercado.jar*.



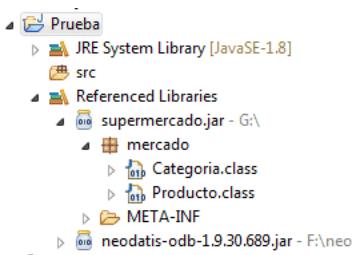
3

Tendrás que acceder a las propiedades del proyecto, concretamente al apartado "Java Build Path" para importar la librería *supermercado.jar* y también la *librería de clases de NeoDatis*.



4

Despliega la librería *supermercado.jar* para comprobar como está organizada.



Puedes comprobar que contiene las clases *Categoría* y *Producto* empaquetadas dentro del paquete *mercado*.

Ya tienes preparado el proyecto que te servirá de base para poner en práctica los ejemplos de los siguientes apartados.

Consultas con la librería NeoDatis

En este apartado tendrás la oportunidad de ejecutar consultas OQL en tus programas Java con ayuda de la librería de clases de NeoDatis.

En el apartado anterior viste el **formato estándar de OQL**. Ahora comprobarás que NeoDatis tiene su forma particular de implementarlo, tal como ocurre con cada gestor de bases de datos orientadas a objetos.

Cuando no deseamos obtener todos los objetos de una determinada clase almacenados en la base de datos, sino sólo aquellos que cumplan algún criterio o condición *where*, necesitamos definir un objeto de la clase *CriteriaQuery* que se construye con dos argumentos: el primero será la clase a la que pertenecen los objetos que deseamos obtener, y el segundo la expresión condicional que determinará los objetos que se van a recuperar. Veamos un ejemplo:

```
CriteriaQuery consulta = new CriteriaQuery(Categoría.class, Where.equal("nombre", "Pescado/Marisco"));
```

Nuestro objeto *consulta* de la clase *CriteriaQuery* representa una condición de búsqueda sobre los objetos de la clase *Categoría*, cuya condición es que el atributo *nombre* sea igual a *Pescado/Marisco*. Este objeto puede ser pasado como argumento al método *getObject()* de la clase *ODB*.

```
Objects<Categoría> categorías = objCategoría.getObjects(consulta);
```

El ejemplo completo quedaría así:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objCategoria = ODBFactory.open("G:/ALMACEN.DB");

        CriteriaQuery consulta = new CriteriaQuery(Categoría.class, Where.equal("nombre", "Pescado/Marisco"));
        Objects<Categoría> categorías = objCategoria.getObjects(consulta);

        Categoría cat;
        while (categorias.hasNext()) {
            cat=categorías.next();
            System.out.println(cat.getNombre());
            for (Producto p : cat.getProductos()) {
                System.out.println(" " + p.getNombre());
            }
        }

        objCategoria.close();
    }
}
```

Para cada objeto *Categoría* obtenido, recorremos sus productos a partir del método *getProductos()*.

También podemos dividir el proceso en dos procesos más pequeños, creando un objeto *ICriterion* en el que se especifica el criterio de búsqueda. Luego, pasamos ese objeto como segundo argumento de nuestro *CriteriaQuery*. El proceso quedaría así:

```
ICriterion criterio = Where.equal("nombre", "Pescado/Marisco");
CriteriaQuery consulta = new CriteriaQuery(Categoría.class, criterio);
Objects<Categoría> categorías = objCategoria.getObjects(consulta);
```

De esta forma, quedará más claro cuando los criterios de búsqueda comienzan a complicarse de verdad.

Presta atención ahora a la expresión siguiente:

Where.equal("nombre", "Pescado/Marisco")

Where es una clase con métodos estáticos que sirven para definir expresiones condicionales, devolviendo un objeto de tipo *ICriterion*. Podemos utilizar uno de los siguientes métodos:

- `Where.equals("atributo",valor);`
El atributo debe ser igual al valor.
- `Where.gt("atributo","valor");`
El atributo debe ser mayor que el valor.
- `Where.ge("atributo","valor");`
El atributo debe ser mayor o igual que el valor.
- `Where.lt("atributo","valor");`
El atributo debe ser menor que el valor.
- `Where.le("atributo","valor");`
El atributo debe ser menor o igual que el valor.
- `Where.not("atributo","valor");`
El atributo debe ser distinto que el valor.
- `Where.isNull("atributo");`
El atributo tiene que ser nulo.
- `Where.isNotNull("atributo");`
El atributo no tiene que ser nulo.
- `Where.like("atributo","patrón de búsqueda");`
Permite especificar un patrón de búsqueda.
Ejemplo:
`ICriterion criterio = Where.like("nombre","Queso%");`
Productos cuyo nombre comienza con la palabra *Queso* seguida de cualquier combinación de caracteres.

En los siguientes apartados realizaremos
más programas de ejemplo.

Ejemplos resueltos

1

Productos que cuestan más de 70 euros

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        ICriterion criterio = Where.gt("precio", 70);
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

2

Productos cuyo nombre comienza con la letra A

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        ICriterion criterio = Where.like("nombre", "A%");
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

Ordenar los resultados

La clase *CriteriaQuery* cuenta con dos métodos que permiten **obtener los resultados de la consulta ordenados de forma ascendente o descendente**.

Se trata de los métodos *orderByAsc()* y *orderByDesc()*. El siguiente ejemplo muestra los productos que cuestan más de 50 euros y los ordena descendientemente, según el valor del atributo *precio*.

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        ICriterion criterio = Where.gt("precio", 50);
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
        consulta.orderByDesc("precio");
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

La clase And

Ya sabemos ejecutar consultas condicionales. Pero, ¿qué pasa cuando queremos expresar criterios más complejos? ¿Y si quiero que se cumplan dos o más condiciones?

La respuesta está en la clase *And*.

Cada objeto de la clase *And* representa una expresión condicional compuesta por una colección de criterios (objetos *ICriterion*), que deberán cumplirse todos para que dicha expresión condicional sea evaluada como verdadera. Una vez configurado correctamente, un objeto de la clase *And* puede ser pasado como argumento al constructor de la clase *CriteriaQuery*.

Veamos un ejemplo:

```
And criterio = new And();
criterio.add(Where.ge("precio", 10));
criterio.add(Where.le("precio", 50));
CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
```

Obtenemos los productos cuyo *precio* esté comprendido entre 10 y 50 euros, es decir, precio mayor o igual a 10, y precio menor o igual a 50.

En el ejemplo puedes ver más claramente que se trata de una colección de objetos *ICriterion*. Una vez construido el objeto *And*, podemos utilizar el método *add()* para añadir tantos objetos *ICriterion* (es decir, condiciones) como sea necesario.

La clase Or

¿Y si deseamos establecer un conjunto de criterios, de manera que sólo deba cumplirse uno de ellos para cada objeto? En ese caso, **utilizaremos la clase *Or***.

Por ejemplo: queremos mostrar los productos cuyo nombre empiece por A, Q o S. Quedaría resuelto así:

```
Or criterio = new Or();
criterio.add(Where.like("nombre", "A%"));
criterio.add(Where.like("nombre", "Q%"));
criterio.add(Where.like("nombre", "S%"));
CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
```

La clase *Or* funciona exactamente igual que la clase *And*; la diferencia está en que para que la expresión condicional sea verdadera, basta con que se cumpla cualquiera de las condiciones de la colección.

Y, ahora, el programa completo:

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

import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        //And criterio = new And();
        //criterio.add(Where.ge("precio", 10));
        //criterio.add(Where.le("precio", 50));

        Or criterio = new Or();
        criterio.add(Where.like("nombre", "A%"));
        criterio.add(Where.like("nombre", "Q%"));
        criterio.add(Where.like("nombre", "S%"));
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);

        consulta.orderByAsc("nombre");
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

Combina And y Or

Los objetos de las clases *And* y *Or* no dejan de ser objetos *ICriterion*, pero más complejos. Realmente, *ICriterion* es una interfaz y existen varias clases que implementan dicha interfaz y que, por lo tanto, también son clases *ICriterion*.

A un objeto de la clase *And* podemos añadirle un objeto *Or*, y viceversa. De este modo, podemos construir criterios mucho más complejos.

Veamos un ejemplo:

```
Or criterio1 = new Or();
criterio1.add(Where.like("nombre", "A%"));
criterio1.add(Where.like("nombre", "Q%"));
criterio1.add(Where.like("nombre", "S%"));

And criterio2 = new And();
criterio2.add(Where.ge("precio", 10));
criterio2.add(Where.le("precio", 50));
criterio2.add(criterio1);

CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio2);
```

En el ejemplo, estamos construyendo el objeto *CriteriaQuery* pasando como segundo argumento la variable *criterio2*. Dicha variable expresa una condición de tipo *And*, configurada de tal modo que deben cumplirse tres condiciones: que el *precio* sea mayor o igual a 10, que el *precio* sea menor o igual a 50 y que el *nombre* comience por A, Q o S. La última condición se ha configurado por medio de un objeto *Or*.

Y, ahora, el programa completo:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.And;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        Or criterio1 = new Or();
        criterio1.add(Where.like("nombre", "A%"));
        criterio1.add(Where.like("nombre", "Q%"));
        criterio1.add(Where.like("nombre", "S%"));

        And criterio2 = new And();
        criterio2.add(Where.ge("precio", 10));
        criterio2.add(Where.le("precio", 50));

        criterio2.add(criterio1);

        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio2);

        consulta.orderByAsc("precio");
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- OQL (*Object Query Language*) es un estándar para creación de consultas en bases de datos orientadas a objetos que nace a partir de SQL (*Structured Query Language*).
- OQL todavía está **en vías de desarrollo**, ya que debido a su complejidad ningún creador de software lo ha implementado completamente.
- NeoDatis tiene su forma particular de implementar OQL. La clase principal relacionada con la ejecución de consultas OQL es *CriteriaQuery*, que se construye con dos argumentos: el primero será la clase a la que pertenecen los objetos que deseamos obtener, y el segundo la expresión condicional que determinará los objetos que se van a recuperar.



PROEDUCA