

IES CHAN DO MONTE

C.S. de Desarrollo de Aplicaciones Multiplataforma

## UNIDAD 4 PARTE 2:

# CONSULTAS HIBERNATE

### Índice

---

<b>1.</b>	<i>Consultas</i> .....	2
<b>2.</b>	<i>Lenguaje de consulta HQL</i> .....	3
<b>2.1</b>	Editor de consulta HQL de Netbeans .....	3
<b>2.2</b>	La cláusula FROM.....	3
<b>2.3</b>	Cláusula SELECT .....	5
<b>2.4</b>	Cláusula WHERE .....	5
<b>2.5</b>	Parámetros en las consultas .....	10
<b>2.6</b>	Recuperar los datos de una consulta .....	11
<b>2.6.1</b>	Query.list().....	11
<b>2.6.2</b>	Query.iterate().....	12
<b>2.6.3</b>	Query.uniqueResult();.....	12
<b>2.7</b>	Consultas sobre clase no asociadas .....	12
<b>2.8</b>	La cláusula ORDER BY .....	13
<b>2.9</b>	Métodos de agregado .....	13
<b>2.10</b>	Agrupaciones .....	14
<b>2.11</b>	Subconsultas .....	14
<b>2.12</b>	Consultas HQL de modificación (Update) .....	14
<b>2.13</b>	Consultas HQL de borrado ( Delete).....	15
<b>2.14</b>	Consultas HQL de insercción (Insert ).....	15
<b>2.15</b>	Asociaciones y Joins .....	15

# 1. Consultas

La instancia **Session** también proporciona acceso a la [API de consultas de Hibernate](#). Vamos a poder hacer consultas utilizando los siguientes métodos.

Vamos a hacer una consulta a la tabla Empleado para obtener los empleados con un sueldo mayor que 23000 utilizando diferentes métodos de expresar una consulta

## ■ [session.createQuery\(hqlQuery\):](#)

**HQL** (Hibernate Query Language): Lenguaje de consultas Hibernate orientado a objetos.

Permite realizar consultas similares a SQL, en términos de [clases persistentes y sus propiedades](#).

Ejemplo:

```

    clase persistente      Atributo de la clase Empleado
    ↓                      ↓
Query consulta=sesion.createQuery("from Empleado where salario>23000");
List <Empleado> lista=consulta.list();   ← Método de la clase Query que devuelve una lista
for (Empleado i:lista){
    System.out.println("Empleado :" +i.getNombre() + " " +i.getApellido1() + " "
                      +i.getApellido2() + " sueldo:" +i.getSalario());
}

```

## ■ [session.createCriteria\(entityClass\)](#)

Dentro de **Hibernate** podemos encontrar la [interfaz Criteria](#) (org.hibernate.Criteria -API de Programación para QBC (Query By Criteria)). Esta interfaz nos permite especificar consultas (en base a clases y métodos de estas clases) sobre nuestras entidades definiendo un conjunto de restricciones.

[QBC es una alternativa a HQL](#), que al igual que ésta permite lanzar consultas, pero a diferencia de ella, no es un lenguaje de consultas, sino una [API que permite expresar consultas en términos de objetos](#) (existen objetos para hacer JOINs, restricciones, etc)

El uso de la clase Criteria puede ser muy conveniente cuando tenemos que componer consultas de forma dinámica, por ejemplo con las típicas pantallas de búsqueda, donde el usuario introduce una serie de criterios. En estos casos en vez de ir componiendo un String con el HQL en función de los datos introducidos por el usuario, puede resultar más cómodo usar la clase Criteria.

```

    Restriccion mayor que      Clase en la que se va a hacer la consulta
    ↓                      ↓
List <Empleado> lista;
lista = sesion.createCriteria(Empleado.class)
    .add(Restrictions.gt("salario", new BigDecimal(23000))).list();
for (Empleado i:lista){
    System.out.println("Empleado :" +i.getNombre() + " " +i.getApellido1() + " "
                      +i.getApellido2() + " sueldo:" +i.getSalario());
}

```

## ■ [session.createSQLQuery\(sqlQuery\)](#)

Se puede utilizar sentencias en SQL nativo para expresar consultas de base de datos. Sólo debe de utilizarse cuando las otras opciones no son válidas (por ejemplo, cuando se necesite utilizar una característica propia del SQL nativo de la BD)

```

    Consulta SQL
    ↓
SQLQuery consulta=sesion.createSQLQuery("Select * from Empleado where salario>23000");
consulta.addEntity(Empleado.class);   ← Añadimos la clase a la que pertenece las columnas devueltas
List <Empleado> lista=consulta.list();
for (Empleado i:lista){
    System.out.println("Empleado :" +i.getNombre() + " " +i.getApellido1() + " "
                      +i.getApellido2() + " sueldo:" +i.getSalario());
}

```

## 2. Lenguaje de consulta HQL

Hibernate utiliza un lenguaje de consulta potente (HQL -Hibernate Query Languaje) que se parece a SQL, es decir, se encuentran estructuradas de forma similar al tradicional con las típicas cláusulas SELECT, FROM y WHERE. Sin embargo, comparado con SQL, HQL **es completamente orientado a objetos** y comprende nociones como herencia, polimorfismo y asociación. HQL es una extensión de SQL orientada a objetos.

El (HQL) es el lenguaje de consultas que usa Hibernate para **obtener los objetos desde la base de datos**. Su principal particularidad es que las **consultas se realizan sobre los objetos java**, que forman nuestro modelo de negocio, es decir, las entidades que se persisten en Hibernate. Ésto hace que HQL tenga las siguientes características:

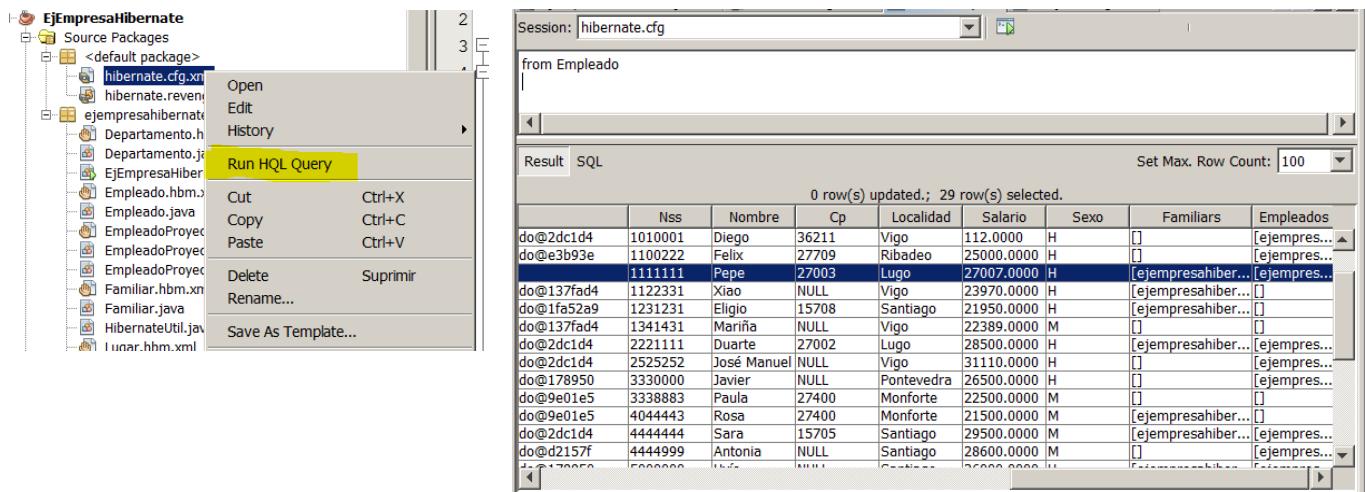
- Los tipos de datos son los de Java.
- Las consultas son independientes del lenguaje de SQL específico de la base de datos
- Las consultas son independientes del modelo de tablas de la base de datos.
- Es posible navegar entre los distintos objetos en la propia consulta.

Las consultas HQL son representadas con una instancia de **org.hibernate.Query**.

Las **consultas no son sensibles a mayúsculas**, a **excepción de los nombres de las clases y propiedades Java**. De modo que WhEre es lo mismo que where igual a WHERE pero org.hibernate.paquete.ANIMAL no es lo mismo que org.hibernate.Paquete.Animal.

### 2.1 Editor de consulta HQL de Netbeans

Netbeans tiene un editor de consultas HQL que ofrece la posibilidad de definir y ejecutar una consulta HQL. Para abrirlo vamos al menú del botón derecho sobre el “hibernate.cfg.xml” y a “Run HQL Query”



### 2.2 La cláusula FROM

**FROM CLASE [AS Alias]**

La cláusula más simple que existe en Hibernate es “**FROM**”. Esta cláusula regresa todas las instancias de la clase indicada que se encuentran en la base de datos, es como si hicieramos un “SELECT \* FROM tabla” en SQL. Por ejemplo, para recuperar todos los departamentos de la base de datos Empresa ejecutamos la consulta de esta forma:

```
from Departamento
```

Porque HQL es orientado a objetos, asume por defecto que seleccionar todas las columnas de cada tabla.

**La cláusula AS:** permite crear un alias para una tabla o un nombre de columna. En general, esta es una buena práctica, especialmente para consultas complejas. Por ejemplo, nuestro anterior ejemplo sencillo sería el siguiente:

```
from Departamento AS D
```

Aunque en muchos casos el uso de alias es opcional, hay situaciones donde es obligatorio.

### Cuándo el uso de alias es opcional

- **Consulta sobre una única entidad sin selección de atributos específicos**

Si queremos recuperar todas las instancias de una entidad sin filtrar ni seleccionar atributos individuales, el alias no es necesario.

```
FROM Departamento
```

- **Consulta con SELECT sobre una sola entidad**

Si se seleccionan atributos de una única entidad, **no es obligatorio** usar alias.

```
SELECT nome FROM Departamento
```

- **Uso en cláusulas WHERE simples**

Cuando solo se consulta sobre una entidad, el alias no es obligatorio:

```
SELECT d.nome FROM Departamento d WHERE d.codig =1
```

También funcionaría sin alias:

```
SELECT nome FROM Departamento WHERE codigo=1
```

### Cuándo el uso de alias es obligatorio

- **Cuando se usa JOIN**

Cuando dos entidades están relacionadas mediante una colección (por ejemplo, Departamento tiene una colección de Empregado), se utiliza JOIN para recuperar información combinada de ambas.

Al realizar **una unión (JOIN)** entre estas entidades relacionadas, es obligatorio asignar un **alias** a cada una.

```
SELECT e.nome, d.nomeDepartamento
FROM Departamento d JOIN d.empregados e
```

- ✓ d.empregados es una colección dentro de Departamento, que representa la lista de empleados asignados a ese departamento.
- ✓ JOIN d.empregados e indica que estamos obteniendo los empleados (e) de cada departamento (d).
- ✓ Como resultado, se generará un registro por cada empleado que pertenece a un departamento.

nome (Empregado)	nomeDepartamento (Departamento)
Ana	INFORMÁTICA
Luis	INFORMÁTICA
Marta	RECURSOS HUMANOS

- **Cuando se usa GROUP BY, ORDER BY o HAVING con SELECT**

Si en una consulta se utiliza **GROUP BY, ORDER BY o HAVING**, los alias son **obligatorios** para referirse a las columnas seleccionadas.

```
SELECT d.nomeDepartamento, COUNT(e)
FROM Departamento d JOIN d.empregados e
GROUP BY d.nomeDepartamento
```

- **Cuando se usan expresiones en SELECT**

Si en la consulta se realizan operaciones, concatenaciones o funciones sobre atributos, se debe usar alias.

```
SELECT CONCAT(e.nome, ' ', e.apellido1) AS nomeCompleto FROM Empregado e
```

- **Cuando se usa CASE**

```
SELECT e.nome,
       CASE WHEN e.salario > 30000 THEN 'Alto'
             ELSE 'Normal'
        END AS Categoría
FROM Empregado
```

## 2.3 Cláusula SELECT

La cláusula SELECT proporciona más control sobre el conjunto de resultados de la cláusula from. Sirve para obtener **algunas propiedades de los objetos** en lugar del objeto completo. A continuación se presenta la sintaxis simple de utilizar la cláusula SELECT para obtener sólo los campos nss y nombre del objeto Empleado:

```
SELECT nss, nome FROM Empleado
```

## 2.4 Cláusula WHERE

La cláusula que permite reducir la lista de instancias devueltas. La cláusula WHERE incluye una comparación, que se utiliza para restringir el número de filas devueltas por la consulta.

Las expresiones utilizadas en la cláusula where incluyen lo siguiente:

### Operadores soportados:

- Matemáticos: +, -, \*, /
- Comparación: =, >=, <=, <>, !=, LIKE
- Lógicos: AND, OR, NOT
- Paréntesis () para agrupar condiciones

En HQL, LIKE solo admite los comodines % (cualquier número de caracteres) y \_ (un solo carácter).

Ej: Seleccionar nss, nombre y apellido1 de los empleados varones que su nombre empiece por J y ganen más de 25000.

```
select nss, nome, apellido1 from Empleado
where salario>25000 and sexo='H' and nome like 'J%'
```

Ej: Encuentra nombres que lleven la palabra ana

```
SELECT e.nome FROM Empregado e WHERE e.nome LIKE '%ana%'
```

- IN / NOT IN: Verifica si un valor está dentro de un conjunto.

Ej: Seleccionar nss, nombre y apellido1 de los empleados que viven en Vigo o Santiago.

```
SELECT e.nss, e.nome FROM Empleado e WHERE e.localidad IN ('Vigo', 'Santiago')
```

- BETWEEN ... AND ...: Comprueba si un valor está en un rango.

Ej: seleccionar nss, nombre y apellido1 de los empleados que su salario esté comprendido entre 25000 y 28000.

```
SELECT e.nss, e.nome FROM Empleado e WHERE e.salario BETWEEN 25000 AND 28000
```

- Case "simple": compara una expresión con un conjunto de expresiones sencillas para determinar el resultado.

**CASE** expresión\_entrada **WHEN** expresión\_when **THEN** resultado\_expresión [...]n  
**[ELSE** resultado\_expresión\_else] **END**

```
SELECT e.nss,
CASE e.tipoContrato
    WHEN 'F' THEN 'Fijo'
    WHEN 'T' THEN 'Temporal'
    ELSE 'Otro'
END
FROM Empleado e
```

Case “de búsqueda”, evalúa un conjunto de expresiones booleanas para determinar el resultado.

**CASE WHEN** expresión\_booleana **THEN** resultado\_expresión [...]n  
**[ELSE** resultado\_expresión\_else] **END**

```
SELECT e.nss,
CASE
    WHEN e.salario > 30000 THEN 'Alto'
    WHEN e.salario BETWEEN 20000 AND 30000 THEN 'Medio'
    ELSE 'Bajo'
END
FROM Empleado e
```

#### ■ Concatenación de cadenas:

Se puede usar **||** o **concat(...)**.

```
select nss, nombre||' '|| apellido1 from Empleado E
```

O usando concat():

```
SELECT e.nss, CONCAT(e.nombre, ' ', e.apellido1) FROM Empleado e
```

#### ■ Funciones de fecha y tiempo

Función	Descripción
<code>current_date()</code>	Devuelve la fecha actual
<code>current_time()</code>	Devuelve la hora actual
<code>current_timestamp()</code>	Devuelve la fecha y hora actual
<code>year(fecha)</code>	Extrae el año de una fecha
<code>month(fecha)</code>	Extrae el mes de una fecha
<code>day(fecha)</code>	Extrae el día de una fecha
<code>hour(fecha)</code>	Extrae la hora de un <code>timestamp</code>
<code>minute(fecha)</code>	Extrae los minutos de un <code>timestamp</code>
<code>second(fecha)</code>	Extrae los segundos de un <code>timestamp</code>

Ej: Obtener empleados contratados este año:

```
SELECT e.nss FROM Empleado e WHERE year(e.fechaContratacion) = year(current_date())
```

### ■ Uso de IS NULL y IS NOT NULL

Para verificar si un campo es nulo o no en una condición WHERE.

Ej: Encuentra empleados que no tienen comisión asignada

```
SELECT e FROM Empleado e WHERE e.comision IS NULL
```

Ej: Encuentra empleados que sí tienen comisión.

```
SELECT e FROM Empleado e WHERE e.comision IS NOT NULL
```

### ■ Uso de COALESCE oisNull

HQL soporta COALESCE() para reemplazar valores NULL por un valor predeterminado. También permite ISNULL(), que internamente se traduce a COALESCE().

```
SELECT e.nombre, COALESCE(e.comision, 0) FROM Empleado e
```

### ■ Manejo de colecciones

- **IS EMPTY / IS NOT EMPTY:** Se usa con colecciones para verificar si están vacías.

```
SELECT e.nss FROM Empleado e WHERE e.departamentos IS NOT EMPTY
```

- **MEMBER OF / NOT MEMBER OF:** Para comprobar si un elemento pertenece a una colección.

Ej: Encuentra los empleados que tienen el número "123456789" en su colección de teléfonos.

```
SELECT e FROM Empregado e WHERE '123456789' MEMBER OF e.telefonos
```

### ■ Contando elementos en HQL

- **size()**

Se utiliza para obtener el número de elementos dentro de una colección. Es una manera práctica de verificar cuántos elementos tiene una colección asociada a una entidad en Hibernate. Se puede utilizar:

#### En la cláusula WHERE:

Podemos usar size() en la cláusula WHERE para filtrar resultados basados en el tamaño de la colección.

En este ejemplo, se seleccionan los empleados que tienen más de un número de teléfono

```
SELECT e FROM Empleado e WHERE size(e.telefonos) > 1
```

#### En la cláusula SELECT:

Podemos usar size() en la cláusula **SELECT junto con GROUP BY** para obtener el tamaño de la colección y otras columnas. En este ejemplo, se selecciona el NSS y el tamaño de la colección de teléfonos para empleados que tienen más de un número de teléfono:

```
SELECT e.nss, size(e.telefonos) FROM Empleado e
WHERE size(e.telefonos) > 1
GROUP BY e.nss
```

- **Como propiedad : colección.size**

Se utiliza para acceder directamente al **tamaño de la colección como una propiedad dentro de la entidad**. Es más intuitivo para los desarrolladores que trabajan con propiedades de objetos Java.

Ejemplo: **e.telefonos.size**

#### En la cláusula WHERE:

Podemos usar e.telefonos.size en la cláusula WHERE para filtrar resultados basados en el tamaño de la colección. En este ejemplo, se seleccionan los empleados que tienen más de un número de teléfono:

```
SELECT e FROM Empleado e
WHERE e.telefonos.size > 1
```

**En la cláusula SELECT:**

Podemos usar `e.telefonos.size` en la cláusula SELECT sin necesidad de GROUP BY. En este ejemplo, se selecciona el NSS y el tamaño de la colección de teléfonos:

```
SELECT e.nss, e.telefonos.size
FROM Empleado e
WHERE e.telefonos.size > 1
```

- **count(t)**

Se utiliza para contar el número de filas que coinciden con una condición específica. Requiere un **JOIN** y **GROUP BY** para contar elementos en una colección.

**En la cláusula WHERE:**

**count()** no se puede usar en la cláusula WHERE, pero podemos usarlo en combinación con HAVING para filtrar grupos basados en el conteo.

**En la cláusula SELECT:**

Podemos **usar count() en la cláusula SELECT** para contar elementos en una colección después de hacer un JOIN. En este ejemplo, se selecciona el NSS y el conteo de teléfonos para empleados que tienen más de un número de teléfono:

```
SELECT e.nss, count(t) FROM Empleado e INNER JOIN e.telefonos t
GROUP BY e.nss
HAVING count(t) > 1
```

Método	Uso en SELECT	Uso en WHERE	Requiere GROUP BY
<code>size(e.telefonos)</code>	Sí	Sí	Sí
<code>e.telefonos.size</code>	Sí	Sí	No
<code>count(t)</code>	Sí (con JOIN)	No	Sí

## ■ Otras funciones de elementos

Función	Descripción	Ejemplo de Uso	Colección Compatible	Salida
<code>minelement()</code>	Devuelve el menor elemento de una colección ordenable.	<code>SELECT p.id,</code> <code>minelement(p.precios) FROM</code> <code>Pedido p</code>	<code>SortedSet</code> , <code>SortedMap</code>	ID del pedido y el menor precio
<code>maxelement()</code>	Devuelve el mayor elemento de una colección ordenable.	<code>SELECT p.id,</code> <code>maxelement(p.precios) FROM</code> <code>Pedido p</code>	<code>SortedSet</code> , <code>SortedMap</code>	ID del pedido y el mayor precio
<code>minindex()</code>	Devuelve el índice más bajo en una colección indexada.	<code>SELECT e.nombre,</code> <code>minindex(e.proyectos) FROM</code> <code>Empleado e</code>	<code>List</code> , <code>Map</code>	Nombre del empleado y el índice más bajo
<code>maxindex()</code>	Devuelve el índice más alto en una colección indexada.	<code>SELECT e.nombre,</code> <code>maxindex(e.proyectos) FROM</code> <code>Empleado e</code>	<code>List</code> , <code>Map</code>	Nombre del empleado y el índice más alto

Este ajuste deja más claro que `minelement()` y `maxelement()` no funcionan con `List` o `Map`, ya que requieren un orden natural en la colección.

## ■ Funciones matemáticas

Función	Descripción	Ejemplo de Uso
<code>abs()</code>	Devuelve el valor absoluto de un número.	<code>SELECT abs(e.salario) FROM Empleado e</code>
<code>sqrt()</code>	Devuelve la raíz cuadrada de un número.	<code>SELECT sqrt(e.salario) FROM Empleado e</code>
<code>mod()</code>	Devuelve el resto de la división de dos números.	<code>SELECT mod(e.salario, 2) FROM Empleado e</code>

## ■ Funciones conversión y extracción

Función	Descripción	Ejemplo de Uso
<code>cast(... as ...)</code>	Convierte un valor a un tipo de Hibernate específico.	<code>SELECT cast(e.salario as string) FROM Empleado e</code>
<code>extract(... from ...)</code>	Extrae una parte específica de una fecha o tiempo: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND.	<code>SELECT extract(year from e.fechaContratacion) FROM Empleado e</code>

## ■ Funciones de cadena

Función	Descripción	Ejemplo de Uso
<code>substring()</code>	Devuelve una subcadena de una cadena.	<code>SELECT substring(e.nombre, 1, 3) FROM Empleado e</code>
<code>trim()</code>	Elimina los espacios en blanco iniciales y finales.	<code>SELECT trim(e.nombre) FROM Empleado e</code>
<code>lower()</code>	Convierte una cadena a minúsculas.	<code>SELECT lower(e.nombre) FROM Empleado e</code>
<code>upper()</code>	Convierte una cadena a mayúsculas.	<code>SELECT upper(e.nombre) FROM Empleado e</code>
<code>length()</code>	Devuelve la longitud de una cadena.	<code>SELECT e.nombre, length(e.nombre) FROM Empleado e</code>
<code>locate()</code>	Devuelve la posición de una subcadena dentro de una cadena.	<code>SELECT locate('a', e.nombre) FROM Empleado e</code>
<code>bit_length()</code>	Devuelve la longitud en bits de una cadena.	<code>SELECT bit_length(e.nombre) FROM Empleado e</code>
<code>str()</code>	Convierte valores numéricos a una cadena.	<code>SELECT str(e.salario) FROM Empleado e</code>

## ■ Funciones especiales elements() e indices

- **elements():**devuelve los valores almacenados en una colección.
  - **En una List o Set:** Devuelve los elementos de la colección.
  - **En un Map:** Devuelve los valores (V) asociados con cada clave (K).
- **indices():** devuelve las claves de una colección indexada.
  - **En una List:** Devuelve los índices de la lista (0, 1, 2...).
  - **En un Map:** Devuelve las claves (K).

**Ejemplo:**

La entidad Empleado tiene un Map<String, String> de teléfonos, donde las claves son los números de teléfono y los valores son la información asociada como "fijo", "móvil", "casa", etc.

```
@Entity
public class Empleado {
    @Id
    private String nss;
    @ElementCollection
    @MapKeyColumn(name = "telefono")
    @Column(name = "informacion")
    private Map<String, String> telefonos = new HashMap<>();
```

Consulta HQL

```
SELECT e.nss, indices(e.telefonos), elements(e.telefonos) FROM Empleado e
```

Salida

```
0010010 111111121 FIJO TRABAJO
0010010 111111122 FIJO TRABAJO
0010010 111111156 FIJO TRABAJO
0010010 111111121 MOVIL PARTICULAR
```

## 2.5 Parámetros en las consultas

Con **CreateQuery** se pueden enlazar valores a los parámetros con nombre, que son identificadores de la forma **:nombre** en la cadena de consulta, o a los parámetros **?** de estilo JDBC. Hibernate numera los parámetros desde 0.

Para asignar los valores a los parámetros se utilizan los métodos setXXX donde XXX es el tipo de datos, por ejemplo:

- **setString(int position o String nome, char val)**
- **setCharacter(int position o String nome, char val)**
- **setBoolean(int position o String nome, char val)**
- **setShort(int position o String nome, char val)**
- **setInteger(int position o String nome, char val)**
- **setLong(int position o String nome, char val)**
- **setFloat(int position o String nome, char val)**
- **setDouble(int position o String nome, char val)**
- **setDate(int position o String nome, char val)**
- **setTime(int position o String nome, char val)**
- **setBinary(int position o String nome, char val)**

### ■ Parámetros posicionales ?

Se basa en definir cada parámetro como un interrogante **"?"**. Posteriormente estableceremos a la clase Query el valor de cada uno de los parámetros.

```
String nombre="ISIDRO";
String apellido1="CORTINA";
String apellido2="GARCIA";

Query query = session.createQuery("SELECT p FROM Empleado p where nombre=? AND
                                   apellido1=? AND apellido2=?");
query.setString(0,nombre);
query.setString(1,apellido1);
query.setString(2,apellido2);
List<Empleado> Empleados = query.list();
for (Empleado Empleado : Empleados) {
    System.out.println(Empleado.toString());
```

## ■ Parámetros por Nombre

En este caso los parámetros se definen como nombre precedidos de dos puntos ":". Esto hace que el código sea más legible y menos propenso a error.

```
String nombre="Oscar";
String apellido1="Galán";
String apellido2="Galán";

Query query = session.createQuery("SELECT p FROM Empleado p where nombre=:nombre AND
                                   apellido1=:apellido1 AND apellido2=:apellido2");
query.setString("nombre",nombre);
query.setString("apellido1",apellido1);
query.setString("apellido2",apellido2);
List<Empleado> Empleados = query.list();
for (Empleado Empleado : Empleados) {
    System.out.println(Empleado.toString());
}
```

## ■ setParameter

Hemos visto cómo el método que usamos para establecer el valor de un parámetro es específico del tipo de dicho parámetro. Esto hace que tengamos que tenerlo en cuenta al escribir el código con la molestia que ello conlleva.

Para mejorar este hecho existe otra forma de asignar valores a los parámetros que es independiente del tipo. Según si el parámetro es por índice o por nombre.

Los métodos son:

```
setParameter(int position, Object val)
setParameter(String name, Object val)
```

En ese caso el código quedaría de la siguiente forma:

```
String nombre="Oscar";
String apellido1="Galán";
String apellido2="Galán";
Query query = session.createQuery("SELECT p FROM Empleado p where nombre=:nombre AND
                                   apellido1=:apellido1 AND apellido2=:apellido2");
query.setParameter("nombre",nombre);
query.setParameter("apellido1",apellido1);
query.setParameter("apellido2",apellido2);
List<Empleado> Empleados = query.list();
for (Empleado Empleado : Empleados) {
    System.out.println(Empleado.toString());
```

## 2.6 Recuperar los datos de una consulta

Hibernate provee métodos para obtener los resultados de un Query: **list()** , **iterate()** y **uniqueResult()**;

### 2.6.1 Query.list()

El método **Query.list()** devuelve en una colección todos los resultados del Query. Es decir, en la colección se encuentran instanciadas todas las entidades que corresponden al resultado de la ejecución del HQL, *por lo tanto el ResultSet subyacente de la conexión JDBC se cierra ya que no se necesita más*.

*Nota: esta alternativa realiza una única comunicación con la base de datos en donde se traen todos los resultados y requiere que haya memoria suficiente para almacenar todos los objetos resultantes del query.*

Ejemplo:

```
Query q=session.createQuery("From Departamento");
List <Departamento> dep=q.list();
```

## 2.6.2 Query.iterate()

El método **Query.iterate()** devuelve un iterador **java.util.Iterator** para iterar los resultados del query. En este caso, las entidades se instancian "a demanda", es decir, con cada llamada al método **Iterator.next()** que devolverá la entidad de la posición actual del iterador.

Para resolver esto, Hibernate ejecuta el query obteniendo sólo los ids de las entidades, y en cada llamada al método **Iterator.next()** ejecuta la consulta propia para obtener la entidad completa. Esto se traduce en una **mayor cantidad de accesos a la base de datos**, resultando en un mayor tiempo de procesamiento total. La ventaja de este método es que no se requiere que todas las entidades estén cargadas en memoria simultáneamente, consumiendo memoria innecesariamente.

Para optimizar la obtención de entidades haciendo que no se realice un acceso a la base por cada llamada al metodo **Iterator.next()**, se puede fijar la cantidad de resultados a traer en cada acceso o "fetch". Esto se configura llamando al método **Query.setFetchSize(n)** antes de la ejecución de la consulta.

Ejemplo:

```
Query q = sesion.createQuery("From Departamento");
q.setFetchSize(10);           // se obtendrán 10 resultados en cada acceso a la base.
Iterator iter = q.iterate();
while (iter.hasNext()) {
    Departamento dep=(Departamento) iter.next();
    .....
}
```

*Nota: Debe tenerse particular cuidado en que no se cierre la sesión de hibernate mientras se utiliza el iterador, de lo contrario no se podrán obtener los resultados subsiguientes de la base de datos.*

### Conclusión

Para los casos en que se deban tener acceso simultáneo a todas las entidades resultantes de un Query, el método **Query.list()** cumplirá con los requisitos, teniendo en cuenta que todos los resultados esperados ocuparán espacio en memoria.

Si lo que se necesita es realizar una operación con cada entidad y luego esa entidad no se utilizará más, el método **Query.iterate()** se ajusta mejor ya que permite limitar (con **Query.setFetchSize()**) la cantidad de resultados que se tienen en memoria y no consumir recursos innecesariamente, haciendo que la aplicación sea más eficiente y menos propensa a fallos por falta de memoria en casos potenciales de muchos resultados en un query.

## 2.6.3 Query.uniqueResult();

Se utiza cuando la consulta devuelve un solo objeto.

```
Query q = sesion.createQuery("From Departamento d where d.nombreDepartamento=?");
q.setString(0, Nombre);
Departamento d= (Departamento) q.uniqueResult();
```

## 2.7 Consultas sobre clase no asociadas

Al recuperar los datos de una consulta en la que intervienen varias tablas y no hay ninguna clase asociada se puede utilizar la clase **Object**. Los resultados se reciben en un array de objetos, donde el primer array se corresponde con la primera clase que ponemos a la derecha del FROM, el siguiente elemento se corresponde con la siguiente y así sucesivamente.

```
Query q = sesion.createQuery("From Departamento d inner join d.empleados order by
d.numDepartamento");
List <Object[]> empd=q.list();
Iterator i=empd.iterator();
while (i.hasNext()){
    Object[] ed=(Object[]) i.next();
    Departamento d=(Departamento) ed[0];
    Empleado e=(Empleado) ed[1];
    System.out.println(d.getNumDepartamento()+" "+e.getNss());
}
```

Salida

```
1 0010010
1 0110010
1 1010001
1 1111111
1 9990009
2 1122331
2 1341431
2 2525252
3 3330000
3 3338883
3 4044443
3 5000000
3 5555000
```

Otra forma de recuperar valores de una consulta que no está asociada a ninguna clase es recuperando una lista de valores de objetos de tipos **java.util.Map**, en donde las claves son las columnas que queremos consultar y los valores son los datos recuperados de la base de datos.

```
Query q = sesion.createQuery("select new map (departamento.numDepartamento , departamento.nombreDepartamento, count(*)) from Empleado "
+ "group by departamento.numDepartamento, departamento.nombreDepartamento");
List <Map> lista=q.list();
for (int i=0;i<lista.size();i++){
    Map map=lista.get(i);
    Set columnas=map.keySet();
    for (Iterator <String> it=columnas.iterator();it.hasNext();){
        String col=it.next();
        System.out.print(map.get(col)+" \t");
    }
    System.out.print("\n");
}
```

Salida:

```
5     PERSONAL      1
3     CONTABILIDAD   2
9     TÉCNICO       3
4     INFORMÁTICA   4
3     ESTADÍSTICA   5
5     INNOVACIÓN    6
BUILD SUCCESSFUL (total time: 1 second)
```

## 2.8 La cláusula ORDER BY

Sirve para ordenar los resultados de su consulta HQL. Se puede ordenar los resultados por cualquier propiedad de los objetos en el conjunto de resultados ya sea ascendente (ASC) o descendente (DESC). A continuación se presenta la sintaxis simple de utilizar la cláusula ORDER BY:

```
FROM Empleado E ORDER BY nombre ASC, apellido1 DESC
```

## 2.9 Métodos de agregado

HQL soporta una amplia gama de métodos globales, similar a SQL. Funcionan de la misma manera en HQL como en SQL y el siguiente es la lista de las funciones disponibles:

Funciones	Descripción
<b>avg</b> (nombre de la propiedad)	El promedio del valor de una propiedad
<b>count</b> (nombre de la propiedad o *)	El número de veces que una propiedad se produce en los resultados
<b>max</b> (nombre de la propiedad)	El valor máximo de los valores de propiedad
<b>min</b> (nombre de la propiedad)	El valor mínimo de los valores de propiedad
<b>sum</b> (nombre de la propiedad)	La suma total de los valores de propiedad

## 2.10 Agrupaciones

Al igual que en SQL se pueden realizar agrupaciones mediante las palabras **claves GROUP BY y HAVING**

Ej: Visualizar cada nombre de empleado con el número de veces que se repite para aquellos empleados que su nombre se repite más de una vez ordenados por el número de veces

```
SELECT nombre, count(nombre) FROM Empleado p GROUP BY nombre HAVING count(nombre)>1
ORDER BY count(nombre)
```

## 2.11 Subconsultas

HQL también soporta subconsultas como en SQL.

Ej: Visualizar los empleados cuya salario es mayor que la media de todos los salarios.

```
SELECT E.nombre, E.apellidos, E.salario FROM Empleado E WHERE E.salario > (SELECT
AVG(E2.Salario) FROM Empleado E2)
```

En HQL, los operadores **SOME**, **ALL**, **EXISTS**, **ANY** e **IN** permiten realizar cuantificaciones en consultas. Se pueden usar con subconsultas y colecciones para evaluar condiciones sobre los elementos de la colección o los resultados de una subconsulta.

### Funciones que pueden cuantificarse con estos operadores

Operador	Descripción	Funciones Cuantificables	Ejemplo de Uso
<b>SOME</b>	Evalúa si al menos un elemento de la colección/subconsulta cumple la condición.	<code>size()</code> , <code>max()</code> , <code>min()</code> , <code>sum()</code> , <code>count()</code> , <code>avg()</code>	<code>SELECT e FROM Empleado e WHERE e.salario &gt; SOME (SELECT p.salario FROM Proyecto p)</code>
<b>ALL</b>	Evalúa si todos los elementos de la colección/subconsulta cumplen la condición.	<code>size()</code> , <code>max()</code> , <code>min()</code> , <code>sum()</code> , <code>count()</code> , <code>avg()</code>	<code>SELECT e FROM Empleado e WHERE e.salario &gt; ALL (SELECT p.salario FROM Proyecto p)</code>
<b>EXISTS</b>	Verifica si una subconsulta devuelve algún resultado.	Funciona con cualquier función dentro de una subconsulta.	<code>SELECT e FROM Empleado e WHERE EXISTS (SELECT 1 FROM Proyecto p WHERE p.empleado = e)</code>
<b>ANY</b>	Equivalente a <b>SOME</b> , evalúa si al menos un elemento de la colección/subconsulta cumple la condición.	<code>size()</code> , <code>max()</code> , <code>min()</code> , <code>sum()</code> , <code>count()</code> , <code>avg()</code>	<code>SELECT e FROM Empleado e WHERE e.salario &gt; ANY (SELECT p.salario FROM Proyecto p)</code>
<b>IN</b>	Comprueba si un valor está dentro de una colección o resultado de una subconsulta.	<code>elements()</code> , <code>indices()</code> , <code>size()</code> , cualquier subconsulta con <code>SELECT</code>	<code>SELECT e FROM Empleado e WHERE e.departamento.id IN (SELECT d.id FROM Departamento d WHERE d.nombre = 'Ventas')</code>

## 2.12 Consultas HQL de modificación (Update)

Actualizar el salario a 34000 del empleado con nss 727755

```
Query query = session.createQuery("update Empleado set salario = :salarioAct "+
" where NSS = :nssAc");
query.setParameter("SalarioAct", 34000)
query.setParameter("nssAc", 727755);
int result = query.executeUpdate();
```

## 2.13 Consultas HQL de borrado ( Delete)

Borrar el empleado con nss 727755

```
Query query = session.createQuery("delete Empleado where NSS = :nssAC");
query.setParameter("nssAC", 727755)
int result = query.executeUpdate();
```

## 2.14 Consultas HQL de insercción (Insert )

En HQL, **solo INSERT INTO ... SELECT ... es soportada**; HQL solo soporta inserciones de otras tablas.

HQL no soporta INSERT INTO ... VALUES.

Ejemplo: Insertar en la tabla empleado los empleados de la tabla CopiaEmpleados2

```
Query query = session.createQuery("insert into Empleado(nss,nombre,
apellido1,apellido2,salario)" +
"select nss,nombre, apellido1,apellido2,salario from CopiaEmpleados2");
int result = query.executeUpdate();
```

## 2.15 Asociaciones y Joins

En **HQL (Hibernate Query Language)**, las asociaciones entre entidades se pueden gestionar mediante **Joins**, lo que nos permite relacionar tablas y optimizar consultas. Hibernate admite los siguientes tipos de joins:

Tipo de Join	Descripción	Sintaxis en HQL (con Colecciones)
Inner Join ( JOIN )	Devuelve solo los empleados que tienen teléfonos asociados.	FROM Empleado e JOIN e.telefonos t
Left Join ( LEFT JOIN )	Devuelve todos los empleados, aunque no tengan teléfonos. Si no tienen, la parte de teléfono será <code>NULL</code> .	FROM Empleado e LEFT JOIN e.telefonos t
Right Join ( RIGHT JOIN )	Devuelve todos los teléfonos, aunque no tengan empleados asociados.	FROM Empleado e RIGHT JOIN e.telefonos t
Full Join ( FULL JOIN )	Devuelve todos los empleados y todos los teléfonos, aunque no tengan relación entre ellos. (Depende del soporte de la BD).	FROM Empleado e FULL JOIN e.telefonos t
Fetch Join ( JOIN FETCH )	Optimiza la carga de datos cuando la colección es <b>LAZY</b> , evitando el problema de <code>n+1 SELECTs</code> .	FROM Empleado e JOIN FETCH e.telefonos

- En HQL no trabajamos con tablas, sino con colecciones y asociaciones.
- Los joins se hacen sobre los atributos de las entidades, no sobre los nombres de las tablas.
- Usar JOIN FETCH mejora el rendimiento y evita problemas con LAZY.

### Lazy vs. Eager Fetching y el uso de FETCH JOIN

Cuando usamos Hibernate, las relaciones pueden cargarse de dos formas:

- **EAGER** (`fetch = FetchType.EAGER`): Se cargan automáticamente con la entidad principal.
- **LAZY** (`fetch = FetchType.LAZY`): Se cargan solo cuando se accede explícitamente a ellas.

Si una colección es **LAZY**, Hibernate no la cargará automáticamente, lo que puede causar problemas si intentamos acceder a ella fuera de la sesión.

Para solucionar esto, usamos **FETCH JOIN**:

```
FROM Empleado as E JOIN FETCH E.telefonos
```

- Esto sobrescribe el fetch = LAZY y fuerza la carga de los teléfonos en la consulta.
- Si no usamos FETCH, se puede generar el error LazyInitializationException si accedemos a la colección fuera de la sesión.

### El problema de las "n+1" SELECTs

Uno de los mayores problemas de rendimiento en Hibernate es el de las **N+1 SELECTs**. Esto ocurre cuando:

1. Primero se ejecuta una consulta para obtener n registros (por ejemplo, todos los Profesores).
2. Luego se ejecuta una consulta adicional por cada registro (n consultas extra para cargar las colecciones asociadas).

#### Ejemplo del problema "N+1"

```
Query query = session.createQuery("SELECT p FROM Profesor p");
List<Profesor> profesores = query.list();

for (Profesor profesor : profesores) {
    System.out.println(profesor.toString());
    for (CorreoElectronico correoElectronico : profesor.getCorreosElectronicos()) {
        System.out.println("\t" + correoElectronico);
    }
}
```

¿Qué ocurre aquí?

1. Hibernate ejecuta 1 consulta para obtener n profesores.
2. Luego ejecuta n consultas más para obtener los correos electrónicos.
3. Si hay 100 profesores, se ejecutan 101 consultas SQL en total.

⚠ Este problema degrada drásticamente el rendimiento si hay muchas filas.

#### ◆ Solución: Usar LEFT JOIN FETCH

La solución más sencilla es **modificar la consulta** para que Hibernate cargue también los correos electrónicos en una sola consulta.

```
SELECT p FROM Profesor p LEFT JOIN FETCH p.correosElectronicos
```

Esto fuerza la carga de los correos electrónicos en la misma consulta, evitando n+1 SELECTs.

### Joins Implícitos vs. Explícitos en HQL

Exactamente, en HQL existen dos formas de hacer joins entre entidades:

- **Forma Explícita** (usando JOIN)
- **Forma Implícita** (usando la notación de puntos .)

#### ■ Forma Implícita (Sin JOIN)

En esta forma, simplemente navegamos por la relación usando la notación de puntos. **Esto solo funciona para relaciones de tipo @ManyToOne o @OneToOne**, porque esas relaciones devuelven **una única entidad**, no una colección.

Ejemplo:

```

@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nss;

    @ManyToOne
    @JoinColumn(name = "numDepartamento")
    private Departamento departamento;
}

@Entity
public class Departamento {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombreDepartamento;
    private int numDepartamento;
}

```

### Consulta en HQL (Forma Implícita)

```

SELECT e.nss, e.departamento.nombreDepartamento FROM Empleado e
WHERE e.departamento.numDepartamento > 3

```

#### 💡 Explicación:

- e.departamento.nombreDepartamento → Hibernate traduce esto en un INNER JOIN implícito.
- Se permite porque departamento es una **referencia a una entidad**, no una colección.

### ▣ Forma Explícita (JOIN) para Colecciones

Si en lugar de una referencia tenemos **una colección** (ejemplo: Empleado tiene una lista de Telefono), **NO podemos usar la notación de puntos**. En su lugar, hay que usar JOIN explícito.

#### Ejemplo con @OneToMany (Un empleado tiene muchos teléfonos)

```

@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nss;
    @OneToMany(mappedBy = "empleado", fetch = FetchType.LAZY)
    private Set<Telefono> telefonos;
}

@Entity
public class Telefono {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String numero;
    @ManyToOne
    @JoinColumn(name = "nss")
    private Empleado empleado;
}

```

#### ✗ Consulta Incorrecta (Intento de usar la notación de puntos en una colección)

```

SELECT e.nss, e.telefonos.numero FROM Empleado e

```

**⚠ Error:** Hibernate no permite esto porque telefonos es una colección, y no se puede navegar por ella con la notación de puntos.

### Consulta Correcta (JOIN)

```
SELECT e.nss, t.numero FROM Empleado e JOIN e.telefonos t
```

#### 💡 Explicación:

- JOIN e.telefonos t → Se usa para recorrer la colección.
- t.numero → Podemos acceder a los atributos de Telefono sin problema.

## ◆ Resumen

Tipo de Relación	Forma Implícita (Puntos . )	Forma Explícita ( JOIN )
@ManyToOne ( Empleado → Departamento )	<input checked="" type="checkbox"/> Sí se puede usar	<input checked="" type="checkbox"/> También funciona
@OneToOne ( Persona → Pasaporte )	<input checked="" type="checkbox"/> Sí se puede usar	<input checked="" type="checkbox"/> También funciona
@OneToMany ( Empleado → Telefonos )	<input checked="" type="checkbox"/> No se puede usar	<input checked="" type="checkbox"/> Se debe usar JOIN
@ManyToMany ( Estudiante → Cursos )	<input checked="" type="checkbox"/> No se puede usar	<input checked="" type="checkbox"/> Se debe usar JOIN

- ✓ Si la relación es **@ManyToOne o @OneToOne** (devuelve UNA sola entidad), podemos **usar la notación**.
- ✓ Si la relación es **@OneToMany o @ManyToMany** (devuelve UNA COLECCIÓN), **debemos hacer un JOIN explícito**.

Ejemplos de consultas:

Visualizar por cada departamento, el numero de empleados

```
select e.departamento.numDepartamento, count(nss) from Empleado e group by e.departamento.numDepartamento
```

Result		SQL
Column 1	Column 2	0
1	5	
2	3	
3	9	
4	4	
5	3	
6	5	

```
select e.departamento.numDepartamento, e.departamento.nombreDepartamento, count(nss) from Empleado e
group by e.departamento.numDepartamento,e.departamento.nombreDepartamento
```

Result SQL

Column 1	Column 2	Column 3
1	PERSOAL	5
2	CONTABILIDAD	3
3	TÉCNICO	9
4	INFORMÁTICA	4
5	ESTADÍSTICA	3
6	INNOVACIÓN	5

```
select departamento.numDepartamento, departamento.nombreDepartamento, count(*) from Empleado
group by departamento.numDepartamento, departamento.nombreDepartamento
having count(*)>3
```

Result SQL

Column 1	Column 2	Column 3
1	PERSOAL	5
3	TÉCNICO	9
4	INFORMÁTICA	4
6	INNOVACIÓN	5

### Cuenta por departamento el numero de empleados temporales

```
select departamento.numDepartamento, departamento.nombreDepartamento, count(*) from Empleado e
where e.class=Empleadotemporal
group by departamento.numDepartamento, departamento.nombreDepartamento
```

Result SQL

Column 1	Column 2	Column 3
1	PERSOAL	1
2	CONTABILIDAD	1
3	TÉCNICO	7
4	INFORMÁTICA	1
5	ESTADÍSTICA	2
6	INNOVACIÓN	1