UAX FP

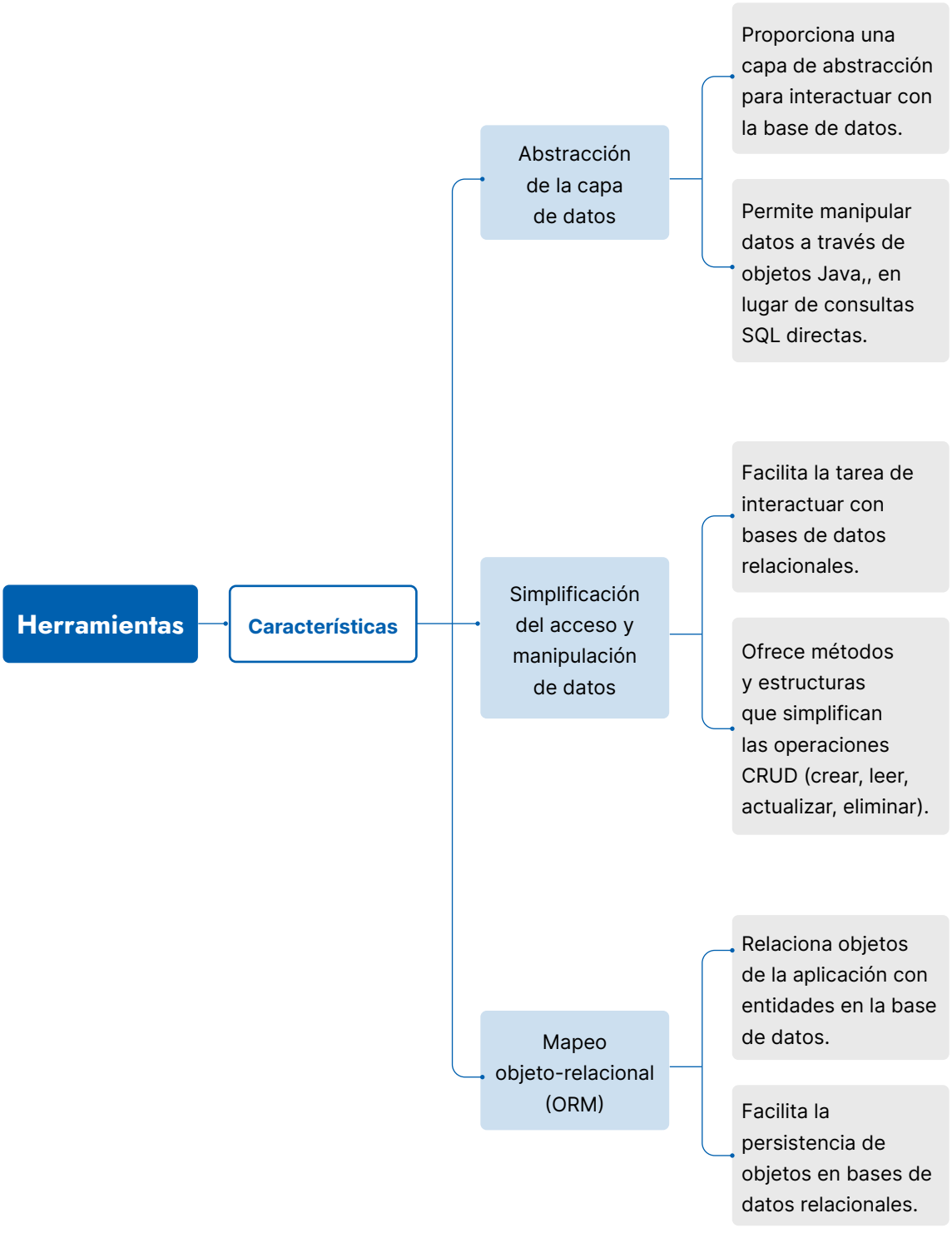
UF3

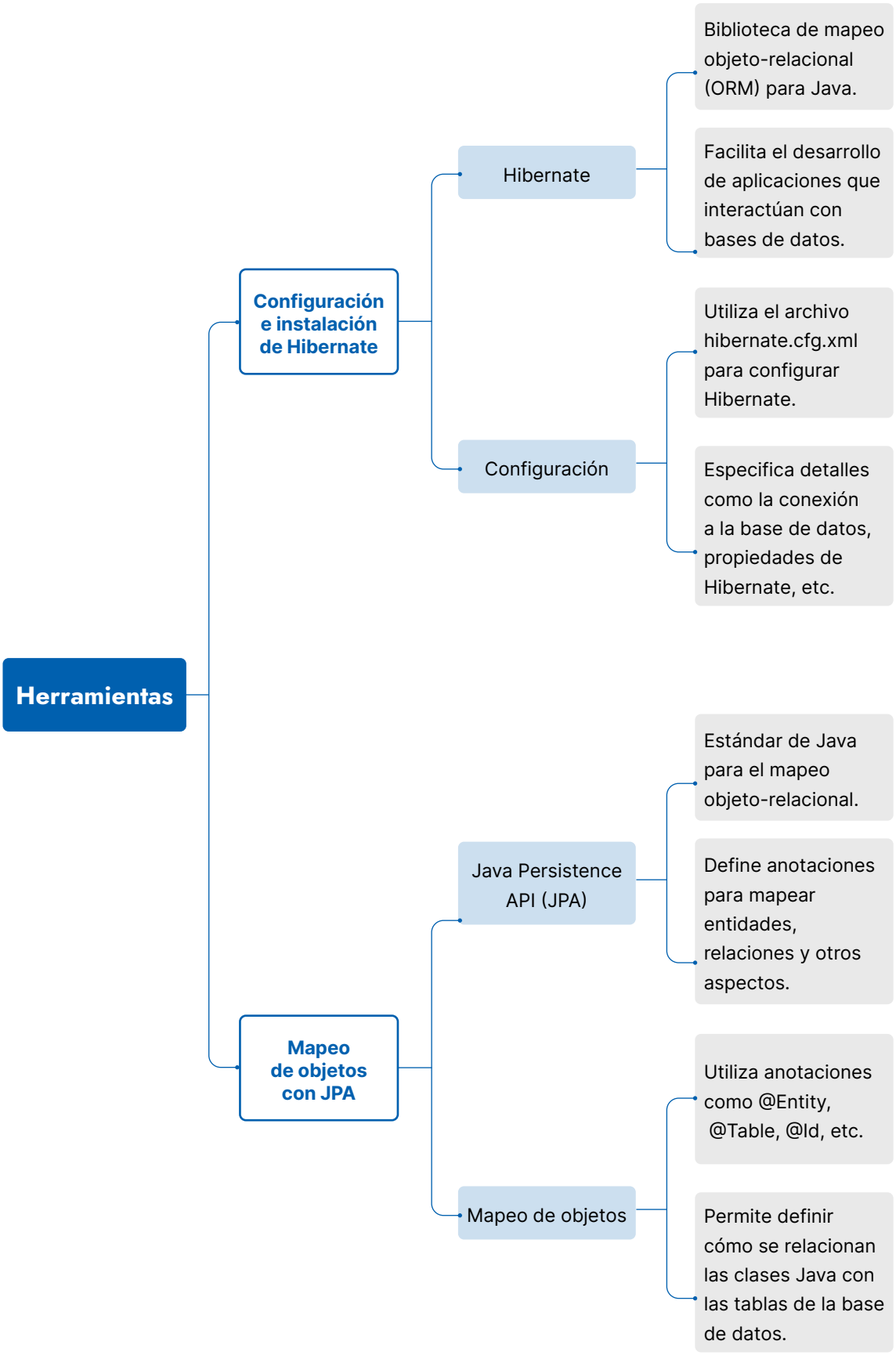
Herramientas de mapeo. Características

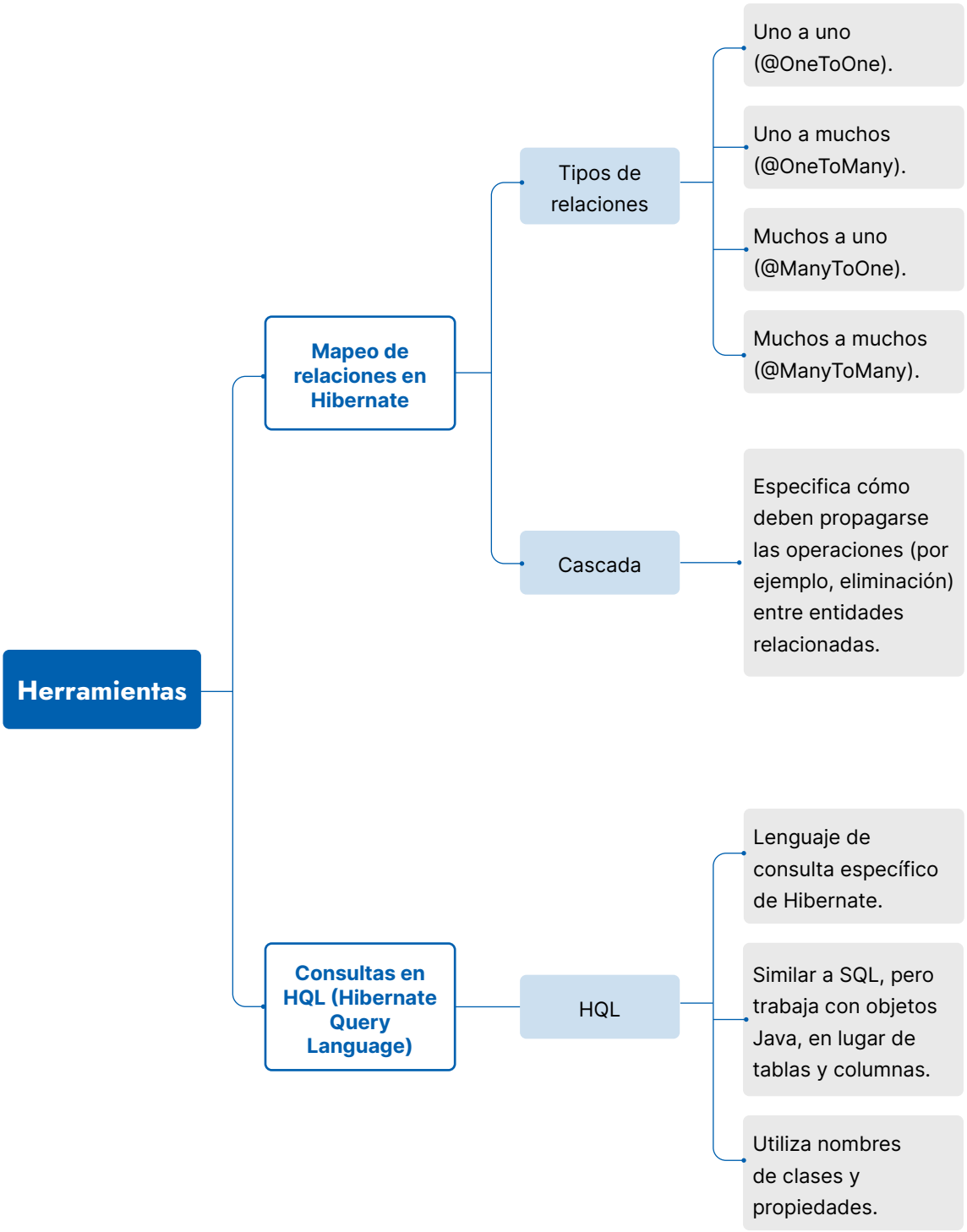
Acceso a datos

ÍNDICE

Mapa conceptual




Herramientas
Configuración e instalación de Hibernate
Hibernate
Biblioteca de mapeo objeto-relacional (ORM) para Java.
Facilita el desarrollo de aplicaciones que interactúan con bases de datos.
Configuración
Utiliza el archivo hibernate.cfg.xml para configurar Hibernate.
Especifica detalles como la conexión a la base de datos, propiedades de Hibernate, etc.
Mapeo de objetos con JPA
Java Persistence API (JPA)
Estándar de Java para el mapeo objeto-relacional.
Define anotaciones para mapear entidades, relaciones y otros aspectos.
Mapeo de objetos
Utiliza anotaciones como @Entity, @Table, @Id, etc.
Permite definir cómo se relacionan las clases Java con las tablas de la base de datos.


Herramientas
Mapeo de relaciones en Hibernate
Tipos de relaciones
Uno a uno (@OneToOne).
Uno a muchos (@OneToMany).
Muchos a uno (@ManyToOne).
Muchos a muchos (@ManyToMany).
Cascada
Especifica cómo deben propagarse las operaciones (por ejemplo, eliminación) entre entidades relacionadas.
Consultas en HQL (Hibernate Query Language)
HQL
Lenguaje de consulta específico de Hibernate.
Similar a SQL, pero trabaja con objetos Java, en lugar de tablas y columnas.
Utiliza nombres de clases y propiedades.

01 Herramientas de mapeo. Características

Vamos a presentar las **herramientas de mapeo** para tratar de solventar el desfase objeto-relacional. Esta no será la única solución, ya que, como veremos en las unidades posteriores, podemos, o bien añadir directamente objetos a nuestras bases de datos relacionales, en las BB. DD. OR (bases de datos objeto-relacionales), o bien trabajar con BB. DD. OO (bases de datos orientadas a objetos).

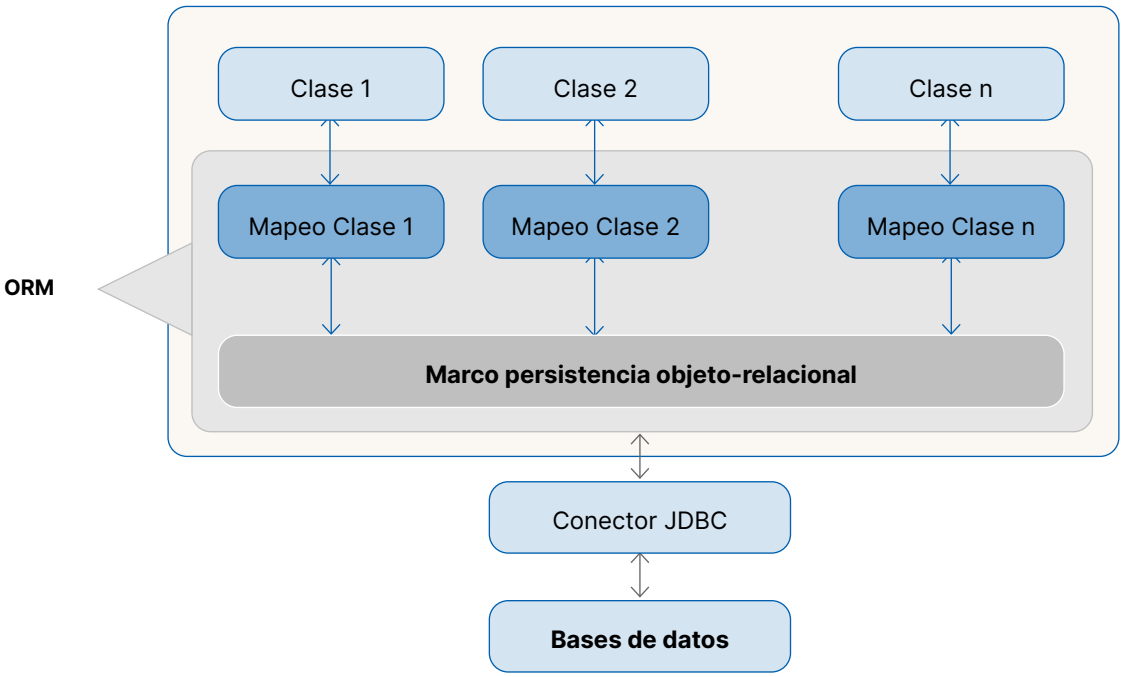
DEFINICIÓN

Las técnicas de los **ORM** (herramientas de mapeo objeto-relacional) se encargan, mediante un conjunto de descripciones y metadatos (datos que describen datos), de realizar una correspondencia entre los datos primitivos de ambos modelos y sus estructuras: entre tablas y objetos, campos y atributos, sus identificadores y sus claves principales.

Esta correspondencia no será siempre sencilla, y habrá que disponer de metadatos que puedan expresar una mayor complejidad. Por ejemplo, nos podemos encontrar con que, a veces, puede interesar almacenar una propiedad en más de una columna, o varias propiedades en una columna única; en otras ocasiones, puede haber propiedades que no se almacenan, o campos de la base de datos que no aparezcan en los objetos; o bien que se utilizan atributos con tipos de datos no primitivos que haya que convertir en otras tablas, y decidir qué campos serán claves externas que apuntan a las nuevas tablas.

IMPORTANTE

Del mismo modo que la definición de los datos, necesitaremos un **mecanismo de persistencia de los objetos,** de manera que los objetos puedan ser «rastreados» en memoria, y que los cambios en estos se vean reflejados directamente en la base de datos.



Esquema 1. Arquitectura de JDBC.

Este tipo de herramientas aportan, entre otras, las siguientes **características**:

- Disminuyen el tiempo de desarrollo.
- Permiten realizar una abstracción del SGBD subyacente, en parte gracias a JDBC.
- Manipularemos solo objetos en nuestro programa, olvidándonos del concepto de tabla, fila y columna.

Cabe destacar que estas ventajas también suponen un coste, el de realizar estas traducciones, que se hacen en cualquier consulta, por lo que el rendimiento no será el mejor.

1.1. Herramientas de mapeo. Características

La mayoría de los lenguajes de programación disponen de herramientas ORM. En estas herramientas ORM hay que distinguir tres componentes:

Técnicas de mapeo	Consistentes en un sistema para expresar la correspondencia entre las clases y el esquema de la base de datos.
Un lenguaje de consulta orientado a objetos	Están formados por una parte Java y por otra que hace uso de bibliotecas del sistema operativo. Su uso se debe a algunos SGBD que incorporan conectores propietarios que no siguen ningún estándar (suelen ser anteriores a ODBC/JDBC).
Técnicas de sincronización	Realmente accederá a las tablas, permitiendo salvar el desfase objeto-relacional.
Tipo IV o Java puros 100 %	Suponen el núcleo funcional para posibilitar la sincronización de los objetos persistentes de la aplicación con la base de datos.

A. Técnicas de mapeo

Destacamos dos **técnicas de mapeo objeto-relacional:**

- Aquellas que incrustan las definiciones dentro del código de las clases y están vinculadas al lenguaje, como las macros de C++ o las anotaciones de PHP y Java.
- Aquellas que guardan las definiciones en ficheros independientes del código, generalmente en XML o JSON.

No se trata de técnicas excluyentes; ambas están disponibles en la mayoría de los entornos, y pueden incluso convivir en la misma aplicación.

B. Lenguaje de consulta

DEFINICIÓN

Todos los SGBDR llevan el soporte de SQL, como es obvio. En las herramientas de mapeo, existe lo que se denomina el **OQL *(object query language)***. Este lenguaje tiene muchas similitudes con SQL, pero una gran diferencia, que se basa en objetos y no en tablas. Cada ORM lleva el OQL a su campo de juego, realizando algunas pequeñas variaciones en cuanto a sintaxis.

C. Técnicas de sincronización

La sincronización es uno de los aspectos más delicados de las herramientas ORM. Consiste en procesos complejos que implican técnicas para las siguientes **funcionalidades:**

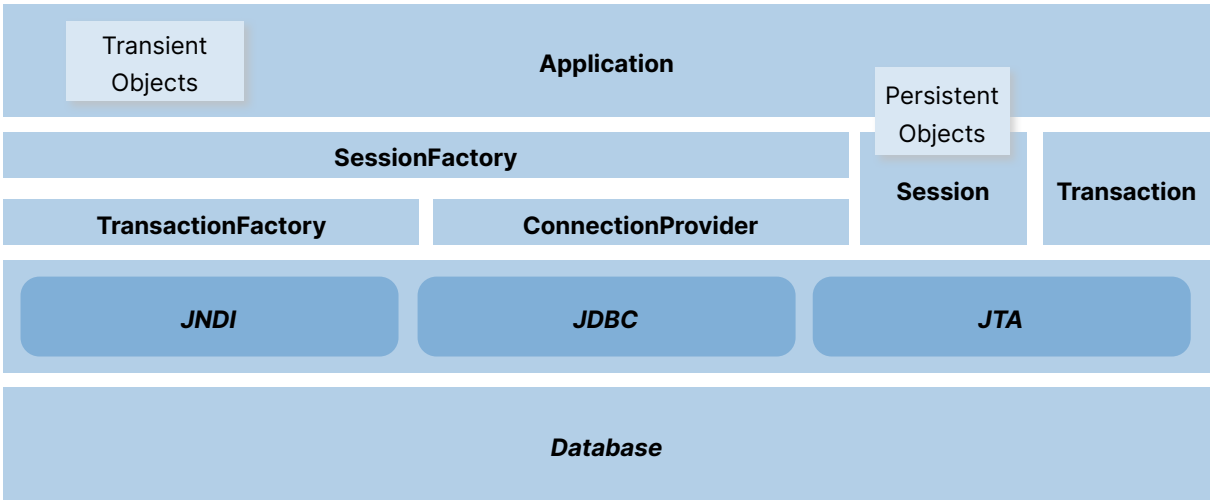
- Descubrir los cambios que sufren los objetos durante su ciclo de vida para poder almacenarlos.
- Crear e iniciar nuevas instancias de objetos a partir de los datos guardados en la base de datos.
- A partir de los objetos, extraer su información para reflejarla en las tablas de la base de datos.

1.2. Hibernate

DEFINICIÓN

Hibernate es un ***framework* ORM** para Java, que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de nuestra aplicación mediante **ficheros XML** o anotaciones en los **beans** de las entidades. Se trata de software libre distribuido con licencia GPL 2.0, por lo que se puede utilizar en aplicaciones comerciales.

La **principal función** de Hibernate será ofrecer al programador las herramientas para detallar su modelo de datos y las relaciones entre ellos, de forma que sea el propio ORM quien interactúa con la base de datos, mientras el desarrollador se dedica a manipular objetos.
Además, ofrece un lenguaje de consulta, denominado **HQL *(Hibernate query language)***, de forma que es el propio ORM quien traduce este lenguaje al propio de cada motor de bases de datos, manteniendo así la portabilidad a expensas de un ligero incremento en el tiempo de ejecución.



Fuente: https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/architecture.html *Arquitectura de Hibernate.*

¿SABÍAS QUE...?

Cuando una aplicación crea objetos, estos están almacenados en memoria, con la volatilidad que esto implica. Dichos objetos se denominan transitorios o `transient`. Con Hibernate, los objetos que tenemos que persistir se «rastrean» en lo que se conoce como una sesión (`Session`), creada a partir de un `SessionFactory`, de acuerdo con la configuración proporcionada. También se proporciona la gestión de conexiones y transacciones.

Como es lógico, los objetos volátiles podrán persistirse, pasando de unos estados a otros, como veremos más adelante, con métodos como `save()` o `persist()`. También se proporciona una interfaz **query**, para lanzar consultas en **HQL** (***Hibernate query language***, su versión del **OQL**).

La parte subyacente de la sesión, como se observa, permite utilizar varias tecnologías, entre ellas **JDBC**, para conectarse a los SGBD necesarios.

02 Configuración e instalación de Hibernate

Para empezar a utilizar Hibernate no hace falta realizar una compleja tarea de descarga de componentes y demás, ya que Hibernate, como buen **framework**, se integrará en nuestro proyecto en forma de librerías. Podemos optar por descargar las librerías en formato **JAR** y añadirlas a nuestro proyecto, pero esto supone que cada vez que portemos un proyecto debamos recordar añadirlas.
La solución, como siempre, son los **gestores de dependencias**, que permiten automatizar dichas tareas y, en las construcciones del proyecto, en caso de no estar físicamente en el sistema, descargar las librerías y añadir las referencias.

2.1. Proyecto con Hibernate y MySQL

VÍDEO LAB

Para consultar el siguiente vídeo sobre Hibernate en Acceso a datos, escanea el código QR o pulsa aquí.

Supongamos que queremos realizar un proyecto en el cual vamos a utilizar Hibernate como ORM entre nuestro proyecto Java y un SGBDR MySQL. Para ello buscamos las dependencias en el gestor de repositorios de **Maven Central** (los enlaces los tienes en el fichero asociado).

Para trabajar con MySQL e Hibernate, una vez buscadas sus dependencias, añadiremos al fichero correspondiente:

EJEMPLO

Proyecto Maven:

```
proyecto/
├── pom.xml
├── README.md
├── LICENSE
└── src/
```

```
    ├── main/
    │   ├── java/
    │   │   └── com/example/proyecto/
    │   │       ├── App.java
    │   │       └── MiClase.java
    │   └── resources/
    │       ├── application.properties
    │       └── imagen.png
    └── test/
        ├── java/
        │   └── com/example/proyecto/
        │       └── MiClaseTest.java
        └── resources/
            └── test-data.json
```

Proyecto Gradle:

```
proyecto/
├── build.gradle
├── README.md
├── LICENSE
└── src/
    ├── main/
    │   ├── java/
    │   │   └── com/example/proyecto/
    │   │       ├── App.java
    │   │       └── MiClase.java
    │   └── resources/
    │       ├── application.properties
    │       └── imagen.png
    └── test/
        ├── java/
        │   └── com/example/proyecto/
        │       └── MiClaseTest.java
        └── resources/
            └── test-data.json
```

CARPETA	MAVEN	GRADLE	DESCRIPCIÓN
pom.xml	Sí	No	Archivo de configuración principal del proyecto que define las dependencias, la configuración de compilación y otros metadatos.
build.gradle	No	Sí	Archivo de configuración principal del proyecto que define las dependencias, las tareas de compilación y otras configuraciones.
src	Sí	Sí	Carpeta raíz para las fuentes del proyecto.
main	Sí	Sí	Contiene el código fuente principal del proyecto.
java	Sí	Sí	Almacena las clases Java del proyecto.
resources	Sí	Sí	Guarda archivos de recursos como application. properties, archivos de configuración XML y otros recursos estáticos.
test	Sí	Sí	Contiene el código de prueba del proyecto.
java	Sí	Sí	Almacena las clases de prueba Java para las unidades y la integración.
resources	Sí	Sí	Guarda archivos de recursos específicos para las pruebas.
README.md	Sí	Sí	Archivo Markdown que contiene información general del proyecto, como la descripción, las instrucciones de instalación y uso, etc.
LICENSE	Sí	Sí	Archivo que define la licencia del software del proyecto.

2.2. Estructura de un proyecto de Hibernate

Una vez tenemos las dependencias añadidas, necesitaremos ver los tipos de archivos con los que vamos a trabajar. Solo introduciremos algunos de ellos, que se desarrollarán en los siguientes apartados.

Cabe destacar que, aunque sin formar parte del proyecto de programación, tendremos ya creada muy probablemente la base de datos, a la que deberemos adaptar nuestro proyecto.

A Bean (Clases)

Deberemos crear las clases que representan a nuestros objetos. Inicialmente, con las salidas de los ORM, estas clases se denominaban **POJO *(plain old Java object)***.

Los **POJO** son objetos comunes, que no pueden heredar ni implementar clases ni interfaces preestablecidas de Java ni tener anotaciones. Básicamente son clases del tipo: «necesito un libro... creo un libro». Los POJO no tienen ningún requerimiento de acceso a sus atributos, ni de cantidad ni de tipo de constructores.

Como una extensión de los POJO aparecen los **beans**, cápsulas o granos (de café), que son más restrictivos, y tienen las siguientes características:

- Hacer sus atributos privados.
- Implementar la interfaz **serializable**.
- Acceder a los campos mediante **getters** y **setters** públicos.
- Implementar un constructor por defecto.

Por tanto, los **beans** serán los componentes de la capa de acceso a datos, y representarán a las diferentes entidades con que trabaja nuestra aplicación.

B Ficheros de mapeado

Una vez definidas las entidades, necesitaremos el fichero de mapeado para cada **bean**. En dicho mapeado se indica a qué tabla de la base de datos se guardará dicho **bean**, así como con qué columna y tipo de datos debe coincidir cada atributo de este.

Deberá existir pues un fichero de mapeado por cada **bean**. Si el **bean** se llama, por ejemplo, `Empleado.java`, el **bean** asociado de llamará `Empleado.hbm.xml` (**hbm** por **Hibernate mapping**). Hablaremos de los mapeados en el apartado siguiente.

C Configuración de Hibernate

Teniendo los **beans** y los mapeados, deberemos especificar la configuración de Hibernate, del mismo modo que establecíamos una configuración en las conexiones con las bases de datos para JDBC. Esta configuración contendrá muchas más cosas, de las que destacamos las más relevantes para trabajar.

La configuración de Hibernate puede especificarse de varios modos:

1. Por **código**.
2. Por fichero **properties**.
3. Por archivo de configuración **XML**.

El tercero es el más utilizado, y el que realizaremos aquí. Hay que comentar que dicho fichero de configuración debe situarse en la raíz del `CLASSPATH` del proyecto, y su nombre debe ser `hibernate.cfg.xml`.

D Resto de clases y aplicación

Por último, tendremos el **resto de clases del programa**, así como la aplicación o clase principal. En caso de estar diseñando una aplicación con entorno gráfico, estarían las clases de representación de los datos o las vistas. Del mismo modo, en caso de una aplicación web, faltarían los controladores y los servicios.

2.3. Configuración del proyecto

Vamos a ver con detalle la configuración de Hibernate. Básicamente, tenemos que realizar dos operaciones, **configurar el proyecto** y **cargar dicha configuración al ejecutarlo**, configuración que pasamos a desarrollar.

03 Mapeo de objetos. JPA

Hemos visto en el apartado anterior la preparación de un proyecto para conectarse con las bases de datos. Ahora tenemos que empezar a mapear entidades, y necesitamos tener el modelo relacional que vamos a mapear.

EJEMPLO

Para ello, partiremos en este primer ejemplo de un supuesto, con la entidad «Pelis» en una base de datos «Cine».

```
CREATE TABLE `Peli` (
  `idPeli` int(11) NOT NULL AUTO_INCREMENT,
  `titulo` varchar(45) NOT NULL,
  `anyo` varchar(45) NOT NULL,
  `director` varchar(45) NOT NULL,
  PRIMARY KEY (`idPeli`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Suponemos que partimos del proyecto con el que finalizamos el apartado anterior, es decir, un proyecto que tiene las dependencias de MySQL y de Hibernate. También disponemos de la clase **HibernateUtil.java** para la gestión de la sesión.
A continuación, vamos a crear el bean que contendrá las Pelis. Lo crearemos en un nuevo paquete al que llamaremos, en inglés, «**Model**», ya que contendrá los beans que conforman el modelo de datos. Dicha clase quedará:

```
package Model;
import java.io.Serializable;

/**
*
      * @author joange
*/
```

```
public class Peli implements Serializable{
    private Long idPeli;
    private String titulo;
    private int anyo;
    private String elDirector;

    public Peli() {
    }

    public Peli(String titulo, int anyo, String elDirector) {
        this.titulo = titulo;
        this.anyo = anyo;
        this.elDirector = elDirector;
    }
 // eliminamos getters, setters y toString
}
```

3.1. Mapeo de entidades. Archivo de mapeo

Está claro que aún no podemos persistir este tipo de objeto. Para poder persistirlo, debemos crear un archivo externo a la clase, de extensión **`.hbm.xml`** y con el mismo nombre de la clase. La localización del archivo no importa ***a priori***, aunque es buena idea tener las clases del modelo por un lado y los archivos de mapeo por otro. Así pues, crearemos un paquete **ORM** y dentro de él crearemos el archivo **`Peli.hbm.xml`**.
La sintaxis básica de estos archivos la vemos a partir del ejemplo:

EJEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
    DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-
    3.0.dtd">
    <hibernate-mapping>
        <class name="Model.Peli" table="Peli" >
            <id column="idPeli" name="idPeli" type="long">
                <generator class="native"></generator>
             </id>
```

```
        <property name="titulo" type="string"/>
        <property name="anyo" />
        <property column="director" name="elDirector" />
    </class>
</hibernate-mapping>
```

La zona del mapeo viene determinada por la etiqueta **<hibernate-mapping>**, donde tenemos:

- **<class>**: indica el mapeo de una clase cuyo nombre está en el atributo **name**, y que se guardará en la tabla indicada por el atributo **table**. Pasaremos ahora a definir los campos de la clase:
  - **id**: este campo representa aquel campo de la base de datos que es la clave principal:
    a) **column**: indica el nombre que tiene en la base de datos, solo cuando es distinto a...
    b) **name**: el nombre que tiene en la clase.
    c) **type**: el tipo de datos. Aquí solo hace falta ponerlo cuando la correspondencia no es equivalente
    (**String-varchar**, **int-numeric**, **double-decimal**, etc.).
  - **generator**: suele utilizarse cuando dicho campo va a ser generado por la base de datos (se corresponde con **AUTO_INCREMENT**). En los objetos, dicho campo **no** se va a facilitar, y es en el momento de persistir los datos cuando se genera (en la base de datos) y se asigna al objeto (en el programa).
  - **property**: para el resto de campos (que no son claves principales).

3.2. Mapeo de entidades. Anotaciones

Como hemos podido comprobar, el mapeo resulta muy sencillo, y se trata simplemente de indicar las equivalencias. El inconveniente es que requiere que debamos manejar dos ficheros, la clase Java y el fichero de mapeo.

Por eso, podemos «fusionar» dicho mapeo y eliminarlo, y podemos realizar dentro de la clase Java las anotaciones pertinentes, mediante el estándar **JPA (Java persistence API)**. La ventaja es clara, solo manipularemos un fichero, aunque, por el contrario, si deseamos dejar de persistir una clase, nos quedará «emborronada» con las anotaciones.
Veamos cómo quedaría la clase:

```
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Column;
import java.io.Serializable;

@Entity
@Table(name="Peli")
public class Peli_Anotada implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long idPeli;

    @Column
    private String titulo;

    @Column
    private int anyo;

    @Column(name="director")
    private String elDirector;

    // Constructores
    public Peli_Anotada() {
    }

   public Peli_Anotada(String titulo, int anyo, String elDirector) {
        this.titulo = titulo;
        this.anyo = anyo;
        this.elDirector = elDirector;
    }
```

```
    // Getters y Setters
    public Long getIdPeli() {
        return idPeli;
    }

    public void setIdPeli(Long idPeli) {
        this.idPeli = idPeli;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public int getAnyo() {
        return anyo;
    }

    public void setAnyo(int anyo) {
        this.anyo = anyo;
    }

    public String getElDirector() {
        return elDirector;
    }

    public void setElDirector(String elDirector) {
        this.elDirector = elDirector;
    }

    // Otros métodos
    @Override
    public String toString() {
          return "Peli_Anotada [idPeli=" + idPeli + ", titulo=" +
titulo + ", anyo=" + anyo + ", elDirector=" + elDirector + "]";
    }
}
```

Se incluyen:

- Constructores: Un constructor vacío y un constructor que inicializa los campos **`titulo`**, **`anyo`**, y **`elDirector`**.
- Getters y Setters: Métodos para acceder y modificar los valores de los campos.
- Método **`toString()`**: Para proporcionar una representación en forma de cadena de la instancia de la clase.

En este caso, se indica que la nueva clase **`Peli_Anotada`** se comportará como una entidad (**`@Entity`**), que estará guardada en una tabla de nombre «**Peli**» (**`@Table(name="Peli")`**). En el fichero **`hibernate.cfg.xml`** tenemos que cambiar el tipo de mapeo, que será, de este modo,

```
<mapping class="Model.Peli_Anotada" />
```

En el interior de la clase, simplemente indicaremos el campo que es clave principal con (**`@Id`**) y que es autoincremental (**`@GeneratedValue`**). Para el resto de campos que queremos que sean mapeados automáticamente, bastará con indicarlos con **`@Column`**. En el caso del campo elDirector, como el nombre es distinto en la base de datos, hay que indicarlo explícitamente (**`name=director`**). Si algún atributo no posee ni **`@Column`** ni **`@Id`**, no será persistido en la base de datos.

3.3. Componentes (@Embedded)

Una **componente** se da cuando en Java tenemos una clase como tal, pero dicha clase solo tiene existencia o sentido dentro de otra clase, sin tener sentido en ninguna otra clase. Un ejemplo podría ser la puntuación en IMDB de una película, que solo tiene sentido que se guarde para dicha película.

Las componentes sustituyen relaciones uno a uno con restricción de existencia, lo que durante el diseño de bases de datos puede agruparlo todo en una única entidad.

Así pues, mediante anotaciones crearemos una clase y la marcamos como **`@Embeddable`** y, dentro de nuestra clase **`Peli`**, marcamos dicho campo componente como **`@Embedded`**:

```
@Embeddable public class IMDB
      implements Serializable {
    @Column
    private String url;
    @Column
    private double nota;
    @Column
    private long votos;
}
```

```
// dentro de Peli @Embedded
private IMDB imdb;
public Peli_Anotada(String
titulo, int anyo,String
elDirector, IMDB imdb) {
  this.titulo = titulo;   this.
anyo = anyo;
  this.elDirector = elDirector;
this.imdb = imdb;
}
```

Después de modificar con el atributo IMDB, es interesante revisar el ***log*** de Hibernate:

```
Hibernate: alter table Peli add column nota double precision
Hibernate: alter table Peli add column url varchar(255)
Hibernate: alter table Peli add column votos bigint
Hibernate: insert into Peli (anyo, director, nota, url, votos,
titulo) values (?, ?, ?, ?, ?, ?)
Peli_Anotada{idPeli=78, titulo=La terminal, anyo=2004,
elDirector=Steven Spielberg, imdb=IMDB{url=https://www.imdb.com/
title/tt0362227/, nota=7.4, votos=438348}}
```

Hemos observado que los campos de la clase IMDB se han integrado perfectamente dentro de la tabla **`Peli`**, lo que hace que esta anotación sea sumamente útil y eficiente.

04 Mapeo de relaciones en Hibernate

Estudiados ya los mapeos de las entidades, vamos a completarlos con las necesarias relaciones. Antes de empezar a comentar la cardinalidad de las relaciones, tenemos que considerar el sentido de dichas relaciones, y vamos a revisar el concepto de direccionalidad de las relaciones:

Unidireccional

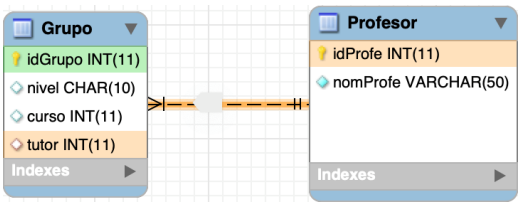
Una relación es unidireccional cuando accederemos al objeto relacionado (componente) a partir de otro objeto (propietario). Por ejemplo, si en un coche montamos un motor, lo lógico es que el propietario es el coche, y, a partir de él, obtendremos el motor.

Bidireccional

Cuando los elementos relacionados suelen tener la misma «ponderación» o entidad. Por ejemplo, un grupo de un instituto y un tutor. A partir de un grupo, sí tiene sentido tener el tutor, y también podemos, a partir de un profesor, acceder al grupo al cual tutoriza.

4.1. Relaciones uno a uno (@OneToOne)

Para la explicación de los ejemplos, veremos el diseño e implementación en la base de datos de cada caso y cómo queda en Hibernate. Para este ejemplo vamos a representar una relación 1:1 entre Grupo y Profesor, donde, como se ve, un grupo posee un tutor.



A. Relación 1:1 unidireccional

En la tabla de **Grupo (columna izquierda)** existe un campo «tutor» del que parte una FK a Profesor. La anotación de `referencedColumnName` es opcional. Por defecto, es la PK de Profesor. Como obligatoriamente un grupo debe poseer un tutor, grupo será la propietaria de la relación (desde grupo «parte» o «sale» la clave ajena). Este ejemplo descrito anteriormente es una relación.

```
@Entity
@Table(name="Grupo")
public class Grupo {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
     @Column(name="idGrupo")
     private Long idGrupo;
     ...
 @OneToOne(cascade=CascadeType.
ALL)
    @JoinColumn(name = "tutor",
referencedColumnName = "idProfe")
     private Profesor elTutor;
     ...
```

```
@Entity
@Table(name="Profesor") public
class Profesor {
    @Id
   @GeneratedValue(strategy =
GenerationType.IDENTITY)     @
Column(name="idProfe")
private Long idProfe;
```

B. Relación 1:1 bidireccional

En el caso de querer **mapearla de manera bidireccional**, simplemente habría que indicar que en la clase Profesor el grupo que queremos que tutorice. El resultado sería el siguiente:

```
@Entity
@Table(name="Grupo")
public class Grupo {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
     @Column(name="idGrupo")
     private Long idGrupo;
     ...
 @OneToOne(cascade=CascadeType.
ALL)
    @JoinColumn(name = "tutor",
referencedColumnName = "idProfe")
     private Profesor elTutor;
     ...
```

```
@Entity @Table(name="Profesor")
public class Profesor {
     @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
     @Column(name="idProfe")
     private Long idProfe;

    @OneToOne(mappedBy =
"elProfe")
    private Grupo elGrupo;
```

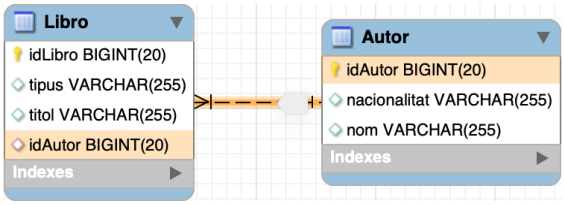
Hemos indicado con este nuevo campo que el propietario de la relación es Grupo.

Algunas opciones de Cascade:

- **`CascadeType.ALL`**: se aplican todos los tipos de cascada.
- **`CascadeType.PERSIST`**: las operaciones de guardado de las entidades propietarias se propagarán a las entidades relacionadas. Solo se aplica si las entidades se guardan con el método **`persist()`** en vez de **`save()`**. Para utilizar **`save()`** con seguridad, reemplaza **`CascadeType.PERSIST`** por **`CascadeType.SAVE_UPDATE`**.
- El resto de opciones, **`CascadeType.MERGE`**, **`CascadeType.REMOVE`**, **`CascadeType.REFRESH`** y **`CascadeType.DETACH`**, realizan lo que su nombre indica, unir, eliminar, refrescar y sacar de la unidad de persistencia. Hay que tener especial cuidado con el **`Remove`**, para evitar borrados en cascada.

4.2. Relaciones uno a muchos (@OneToMany/@ManyToOne)

Para esta explicación partiremos del siguiente modelo, en el cual un Libro posee un Autor que lo ha escrito, y un Autor puede haber escrito varios Libros. En el esquema relacional la relación es de **`idAutor`** en Libros, que es FK hacia la tabla Autor.



A. Relación unidireccional

En este caso, **la unidireccionalidad debe aparecer solo en la tabla propietaria (Libro)**. Así pues, en la clase libro indicaremos que contiene un Autor, mapeándola como se ve en el cuadro a continuación. En la tabla Autor no se indicará nada.

En el campo tipo Autor (objeto) se mapea con la anotación **`@ManyToOne`** (muchos libros escritos por un Autor) a la columna **`idAutor`**. Se indica que cualquier guardado en Libro provoca que se persista también el Autor (**`CascadeType=PERSIST`**). También se anota que la **FK** tendrá como nombre **`FK_LIB_AUT`**. Veamos qué ocurre al ejecutar el siguiente programa.

B. Relación uno a muchos bidireccional

El **mapeado de la relación bidireccional** nos va a permitir acceder a la información relacionada en ambas direcciones. Para ello añadiremos en Autor un conjunto (**`Set`**) de todos los libros que ha escrito.

```
@Entity
@Table(name="Libro")
public class Libro implements
Serializable {
        @Id @GeneratedValue(…)
        private Long idLibro;
           …
@ManyToOne(cascade=CascadeType.
PERSIST)
@JoinColumn(name="idAutor",
foreignKey = @ForeignKey(name =
"FK_LIB_AUT" ))
     private Autor elAutor;
```

```
@Entity
@Table(name="Autor")
public class Autor {
@OneToMany(mappedBy="elAutor",
             cascade=CascadeType.
PERSIST,
            fetch =
FetchType.LAZY)
    private Set<Libro> losLibros;
```

En Autor, como tabla relacionada no propietaria, se indica que la información de la relación está en el campo «**`elAutor`**», dentro de la anotación **`@OneToMany`**. Aparece un nuevo marcador **fetch**, (existente en las colecciones), cuyo comportamiento determinará el momento en que se realiza la carga de los datos de las colecciones, y cuyos valores pueden ser:

FechtType.EAGER	Literalmente, como «modo ansioso». No podemos esperar, y en el momento de la carga del Autor, Hibernate resolverá la relación y cargará todos los libros con todos los datos internos de cada libro. Tenemos todos los datos al momento.
FechtType.LAZY	Carga en diferido; al cargar el Autor, Hibernate solo carga los atributos propios del Autor, sin cargar sus Libros. En el momento en que queramos acceder a dicha información es cuando se realiza la carga de estos. Es decir, en el modo **`LAZY`**, los datos se cargan «cuando hacen falta».

4.3. Relaciones muchos a muchos (@ManyToMany)

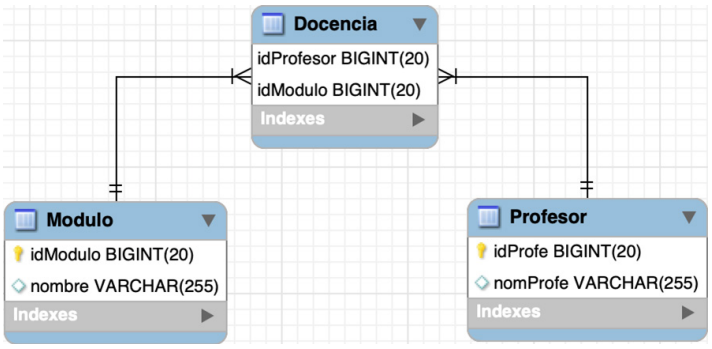
Dentro de las **relaciones binarias,** podemos encontrar dos posibilidades:

- Relaciones que simplemente indican la relación (por ejemplo, que un personaje puede llevar o no tal tipo de arma en un juego de rol).
- Relaciones que, aparte de indicarla, añaden nuevos atributos a esta (un actor participa en una película con un tipo de papel: principal, secundario, etc.).

En el modelo relacional, ambos casos terminaban modelándose como una nueva tabla (con o sin el atributo), pero en el modelado OO:

- **En el primer caso**, no modelamos una clase para mapear dicha tabla de la base de datos.
- **En el segundo caso**, debe modelarse con una clase con el atributo de la relación, por lo que la relación N:M entre dos tablas serán dos relaciones uno a muchos, 1:N y N:1, cosa que ya sabemos resolver.

Vamos a modelar el típico caso de un Profesor que imparte varios Módulos, que pueden ser impartidos por varios profesores. El esquema se ve a continuación:



Como podemos observar, queda la típica tabla central de la relación N:M. Como se ha comentado anteriormente, la tabla de Docencia no existirá en el modelado OO, ya que únicamente sirve para relacionar los elementos.
Las clases `Modulo` y `Profesor` quedan como sigue (se muestra solo la parte relacionada con la relación), eligiendo en este caso `Profesor` como propietaria de la relación.

Profesor

```
@ManyToMany(cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)
@JoinTable(name="Docencia",
            joinColumns = {@JoinColumn(name="idProfesor",
               foreignKey = @ForeignKey(name = "FK_DOC_PROF" ))},
            inverseJoinColumns = {@JoinColumn(name="idModulo",
               foreignKey = @ForeignKey(name = "FK_DOC_MOD" ))})
private Set<Modulo> losModulos=new HashSet<>();;
public void addModulo(Modulo m){
    if (!this.losModulos.contains(m)){
         losModulos.add(m);
    }
    m.addProfesor(this);
}
```

Módulo

```
@ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY,
             mappedBy = "losModulos")
private Set<Profesor> losProfesores=new HashSet<>();;
public void addProfesor(Profesor p) {
   if (!this.losProfesores.contains(p)) {
       losProfesores.add(p);
    }
    p.addModulo(this);
}
```

Fíjate en lo siguiente:

- En ambas clases, el mapeo es **`@ManyToMany`**, conteniendo un **`Set`** de objetos de la otra clase, indicando las operaciones en cascada (**`cascade`**) y la carga de los objetos relacionados (**`fetch`**).
- En la clase propietaria (**`Profesor`**) se inicia la relación, enlazando con una tabla **`Docencia`** siguiendo la FK —desde el origen hasta la punta de la flecha—, donde:
  - Se va a enlazar (**`joincolumns`**) con el campo **`idProfe (@JoinColumn)`**.
- Se mapea desde la tabla **`Docencia`** hasta la entidad de origen **`Modulo`** de manera inversa (de punta al origen de la flecha):

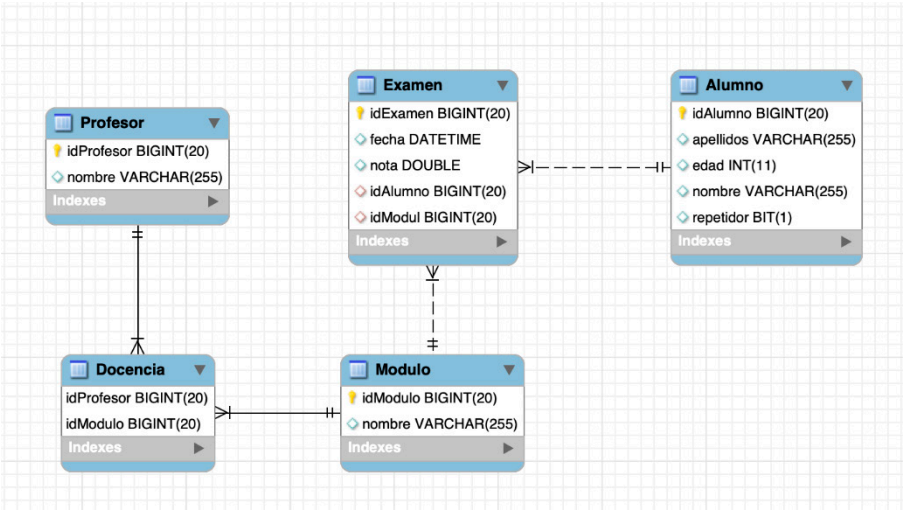
- Esto se consigue con **`inverseJoinColumns`**, enlazando desde el campo **`idModulo (@JoinColumn)`**

» En la clase relacionada (**`Modulo`**), que no es la propietaria, simplemente le indicamos que la propietaria es **`Profesor`**, mediante **`mappedBy="losModulos"`**.

Fíjate en que los métodos **`addModulo`** de **`Profesor`** y **`addProfesor`** de **`Modulo`** se llaman entre sí. Para evitar entrar en una recursión, se ha realizado un control que frene dicha posible casuística. Además, al estar enlazados garantizamos la persistencia en cascada, tal y como hemos programado las relaciones.

05 Consultas con HQL

Para concluir con la unidad, necesitamos ver maneras de recuperar información desde la base de datos. El lenguaje **HQL*(Hibernate query language)*** nació con la finalidad de salvar de nuevo el modelo relacional, ya que es un supraconjunto de SQL (ampliación de SQL). La primera consideración es que, por defecto, su funcionalidad es la de **recuperar objetos de la base de datos**, no tablas, como hacíamos en el lenguaje SQL mediante los **`ResultSet`**. Las consultas con HQL se realizarán a partir de una interfaz **Query**, que será el modo donde especificaremos qué queremos recuperar. Opcionalmente podemos añadir a la consulta los parámetros necesarios para su ejecución, para evitar consultas ***hard-coded***.



Clases y Métodos Relevantes:

1. **HQL (Hibernate Query Language)**:
   - Es un lenguaje de consulta específico de Hibernate, diseñado para realizar consultas a bases de datos relacionales utilizando el modelo de objetos de Hibernate.
   - Funciona como un supraconjunto de SQL, lo que significa que ofrece funcionalidades más avanzadas que SQL estándar.

2. **Query Interface**:
   - Es una interfaz en Hibernate que se utiliza para crear y ejecutar consultas HQL.
   - Permite especificar qué datos se desean recuperar de la base de datos y opcionalmente añadir parámetros para personalizar la consulta.

Conceptos Importantes:

1. **Recuperación de Objetos**:
   - A diferencia de SQL estándar, que recupera filas de tablas como objetos ResultSet, HQL recupera objetos directamente de la base de datos.
   - Esto significa que las consultas HQL devuelven instancias de clases de entidad mapeadas en lugar de simples conjuntos de resultados.

2. **Parámetros de Consulta**:
   - Permiten agregar flexibilidad a las consultas al permitir la parametrización de los criterios de búsqueda.
   - Evitan la escritura de consultas “hard-coded” al permitir que los valores de búsqueda se pasen dinámicamente a la consulta en tiempo de ejecución.

EJEMPLO

```
Query query = session.createQuery("SELECT p FROM Peli_Anotada p WHERE
p.anyo = :anyo");
query.setParameter("anyo", 2022);
List<Peli_Anotada> peliculas = query.list();
```

En este ejemplo, se utiliza la interfaz **`Query`** para crear una consulta HQL que recupera instancias de la clase **`Peli_Anotada`** donde el año sea igual a 2022. Se utiliza un parámetro **`:anyo`** en la consulta para evitar valores hard-coded y luego se establece el valor del parámetro utilizando el método **`setParameter()`**. Finalmente, se ejecuta la consulta y se obtiene una lista de objetos **`Peli_Anotada`** como resultado.

5.1. Recuperación de objetos simples

Estas consultas son las que permiten recuperar un objeto o colección de objetos desde las bases de datos.
Veamos los ejemplos; el primero muestra todos los alumnos:

Código completo con encabezados de métodos, excepciones y comentarios:

```
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.NonUniqueResultException;
import java.util.List;

public class ConsultasHQL {

    /**
     * Recupera todos los alumnos de la base de datos.
     *
     * @param laSesion la sesión de Hibernate para realizar la consulta
     */
    public void recuperarTodosLosAlumnos(Session laSesion) {
        Query query = laSesion.createQuery("SELECT a FROM Alumno a");
        List<Alumno> alumnos = query.list();

        for (Alumno alumno : alumnos) {
            System.out.println(alumno);
        }
    }

    /**
     * Recupera un único alumno de la base de datos mediante su ID.
     *
```

```
     * @param laSesion la sesión de Hibernate para realizar la
consulta
     * @param idAlumno el ID del alumno que se desea recuperar
     * @return el alumno recuperado, o null si no se encuentra
ningún alumno con el ID proporcionado
     */
    public Alumno recuperarAlumnoPorId(Session laSesion, long
idAlumno) {
        Query query = laSesion.createQuery("SELECT a FROM Alumno a
WHERE a.idAlumno = :id");
        query.setParameter("id", idAlumno);

        Alumno alumno = (Alumno) query.uniqueResult();
        return alumno;
    }

    /**
     * Ejemplo de cómo utilizar la paginación en una consulta HQL
para evitar recuperar
     * todos los resultados de una sola vez.
     *
     * @param laSesion la sesión de Hibernate para realizar la
consulta
     * @param inicio el índice de la primera fila a recuperar
     * @param cuantos el número de filas a recuperar
     */
    public void consultaConPaginacion(Session laSesion, int
inicio, int cuantos) {
        Query query = laSesion.createQuery("SELECT a FROM Alumno
a");
        query.setFirstResult(inicio);
        query.setMaxResults(cuantos);

        List<Alumno> alumnos = query.list();
        for (Alumno alumno : alumnos) {
            System.out.println(alumno);
        }
    }
}
```

- **Encabezados de Métodos**: Descripciones detalladas de los métodos para indicar su funcionalidad y cómo se utilizan.
- **Excepciones**: Se menciona la excepción `NonUniqueResultException` que puede ocurrir cuando se utiliza `uniqueResult()` y el resultado no es único.
- **Comentarios**: Se proporcionan comentarios para explicar el propósito de cada método y cómo se utilizan los parámetros.

En formato tabla quizás sea más visual:

MÉTODO	DESCRIPCIÓN
`recuperarTodosLosAlumnos (Session laSesion)`	Recupera todos los alumnos de la base de datos.
`recuperarAlumnoPorId (Session laSesion, long idAlumno)`	Recupera un único alumno de la base de datos mediante su ID.
`consultaConPaginacion(Session laSesion, int inicio, int cuantos)`	Ejemplo de cómo utilizar la paginación en una consulta HQL para evitar recuperar todos los resultados de una sola vez.

Este segundo ejemplo permite obtener solo uno, con lo que se indica en la cláusula **`where`**.

```
Alumno a = (Alumno)q.uniqueResult();
Query<Alumno> q = laSesion.createQuery("Select a from Alumno a where a.idAlumno=1",Alumno.class); Alumno a = q.uniqueResult();
```

En consultas con **`uniqueResult`**, si el resultado es más de uno, aparecerá la excepción **`org.hibernate.NonUniqueResultException`**, ya que el **`where`** no filtra por el identificador.

Cuando los resultados sean muchos, igual no conviene recuperarlos todos de golpe, sino ir accediendo a ellos de diez en diez o de una forma similar, al igual que las páginas de búsqueda de Google o Amazon. Esto lo podemos conseguir lanzando consultas más pequeñas de manera repetida, aplicándole al **query**:

- `Q.setFirstResult`(int inicio): indica la primera fila que devolverá.
- `Q.setMaxResult`(int cuantos): indica cuantas filas devuelve.

Con un algoritmo apropiado, podemos realizar un **bucle**, desplazando en cada iteración el inicio e incrementándolo en la cantidad de filas recuperadas en la anterior iteración. Implicaría muchas consultas pequeñas frente a una grande.

5.2. Consultas mixtas

DEFINICIÓN

Entendemos por **consultas mixtas** aquellas que no devuelven objetos enteros como tales. Estas consultas devolverán, o bien parte de objetos, o bien un objeto más algo más.

La novedad es que el resultado será un array de objetos (`Object[ ]`), y, por lo tanto, deberemos ser muy cuidadosos con el tipo de cada celda, así como con el tamaño de dicho array, ya que estará fuertemente ligado a la propia consulta. Veamos la siguiente consulta, Mostrar nombre y edad de los alumnos:

```
Query q = laSesion.createQuery("Select a.nombre,a.edad from Alumno
a"); List<Object[]> losAlumnos = q.list(); for (Object[] alu :
losAlumnos) {
    System.out.println("El alumno " + alu[0] + " tiene " + alu[1]
+ " años");
}
```

Lo que cambia es la manera de procesar el resultado. Al ser un array de `Object`, deberemos acceder a las columnas como si fuera una tabla. En este ejemplo para mostrar, gracias al polimorfismo de Java, no hace falta nada más, pero sí que sería interesante hacer cásting de `Object` a los tipos necesarios para manipular los datos.

5.3. Los múltiples Select en cascada

Cuando realicemos una consulta a una clase que contiene objetos enlazados, habitualmente por relaciones, está consulta generará una consulta anidada por cada objeto enlazado, para cargar su contenido. Este es el comportamiento con una consulta ansiosa (`Eager`). Habrá que valorar el cargarlas en diferido o `Lazy`. Puedes observar este comportamiento en el archivo externo **Ejemplo1.java**, incluido en el fichero **AD_U03_A05_02.zip**, ubicado en la sección **Ejemplos para trabajar y analizar**.

5.4. Consultas sobre colecciones

Vamos a consultar el nombre de los alumnos y cuántos exámenes ha realizado cada uno. Dicha información está en el set de exámenes, por lo que necesitaremos manipular dicha colección:

```
Query q = laSesion.createQuery("Select a.nombre,size(a.losExamenes)
from
Alumno a");
List<Object[]>  losAlumnos  =
q.list(); for (Object[] alu :
losAlumnos) {
    System.out.println("El alumno " + alu[0] + " ha hecho" + alu[1]
+ " examenes");
}
```

Como puede apreciarse, hemos aplicado la función **`size()`** a la colección para ver su tamaño. Podemos aplicar, por tanto:

- **`Size(colección)`**: recupera el tamaño.
- **`Colección is empty | colección is not empty`**: para determinar si está vacía. Equivale a comparar el **`size`** con 0.
- Pueden combinarse los operadores **`in`**, **`all`** mediante el operador **`elements(colección)`**.

5.5. Consultas con parámetros. Consultas nominales

Normalmente, la mayoría de consultas necesitarán unos parámetros, en general para el filtrado de objetos en la cláusula **`where`**. Ya se comentaron en la unidad anterior las bondades de parametrizar las consultas para evitar problemas de inyección SQL o similares.
La gestión de parámetros se realiza del mismo modo que con las sentencias (**`Statements`**) y puede realizarse mediante parámetros posicionales o nominales.

Para comprender mejor cómo funcionan los parámetros posicionales, nominales y las consultas nominales en Hibernate, podemos proporcionar ejemplos prácticos:

A. Parámetros posicionales:

```
Query query = session.createQuery("SELECT a FROM Alumno a WHERE a.edad
> ? AND a.curso = ?");
query.setParameter(0, 18); // Primer parámetro posicional: edad > 18
query.setParameter(1, "Segundo curso"); // Segundo parámetro posicional:
curso = "Segundo curso"
List<Alumno> alumnos = query.list();
```

B. Parámetros nominales:

```
Query query = session.createQuery("SELECT a FROM Alumno a WHERE a.edad
> :edad AND a.curso = :curso");
query.setParameter("edad", 18); // Parámetro nominal "edad"
query.setParameter("curso", "Segundo curso"); // Parámetro nominal
"curso"
List<Alumno> alumnos = query.list();
```

C. Consultas nominales:

```
// Definición de consulta en la clase de entidad Alumno o en un archivo
de configuración
@NamedQuery(
    name = "buscarAlumnosPorEdadYCurso",
    query = "SELECT a FROM Alumno a WHERE a.edad > :edad AND a.curso =
:curso"
)

// Uso de la consulta nominal en el código
Query query = session.getNamedQuery("buscarAlumnosPorEdadYCurso");
query.setParameter("edad", 18); // Parámetro nominal "edad"
query.setParameter("curso", "Segundo curso"); // Parámetro nominal
"curso"
List<Alumno> alumnos = query.list();
```

Explicación:

TIPO DE PARÁMETRO	EJEMPLO DE USO	DESCRIPCIÓN
Parámetros posicionales	`java Query query = session.createQuery("SELECT a FROM Alumno a WHERE a.edad > ? AND a.curso = ?"); query.setParameter(0, 18); query.setParameter(1, "Segundo curso");`	Se marcan dentro de la consulta con el símbolo de interrogación seguido de un número (?1, ?2, etc.). Luego se asigna el valor correspondiente utilizando el método **`setParameter(posicion, valor)`**.
Parámetros nominales	`java Query query = session.createQuery("SELECT a FROM Alumno a WHERE a.edad > :edad AND a.curso = :curso"); query.setParameter("edad", 18); query.setParameter("curso", "Segundo curso");`	Se indican en la consulta con el nombre del parámetro precedido por dos puntos (:edad, :curso). Luego se asigna el valor correspondiente utilizando el método **`setParameter ("nombreParametro", valor)`**.
Consultas nominales	`java @NamedQuery(name = "buscarAlumnosPorEdadYCurso", query = "SELECT a FROM Alumno a WHERE a.edad > :edad AND a.curso = :curso") Query query = session.getNamedQuery ("buscarAlumnosPorEdadYCurso"); query.setParameter("edad", 18); query.setParameter("curso", "Segundo curso");`	Se definen fuera de la sesión de Hibernate en la clase de entidad o en un archivo de configuración. Luego se invocan en el código mediante **`getNamedQuery ("nombreConsulta")`** y se asignan los parámetros necesarios con **`setParameter( "nombreParametro", valor)`**.

5.6. Inserciones, actualizaciones y borrados

Por último, vamos a analizar el resto de las operaciones CRUD. Hay que comentar que **estas operaciones pueden realizarse directamente sobre los objetos**; por eso se explican con carácter complementario y no principal.

Hacemos notar que, para la inserción, no se permite asignar los valores directamente, sino que solo los podemos obtener de una subconsulta:

```
insert into entidad (propiedades) select_hql
```

La sintaxis de update o delete es similar a la de SQL:

```
update from entidad set [atrib=valor] where condicion
delete from entidad where condicion
```

Además de todo lo visto:

- Estas instrucciones pueden contener parámetros.
- El **`where`** es opcional, pero lo borrará o actualizará todo.
- Estas consultas se ejecutan todas mediante **`executeUpdate()`**, que retorna un entero con el número de filas afectadas.

CLAVES Y CONSEJOS

En el caso de manipulación de objetos (borrados, modificaciones, etc.) tenemos suficientes herramientas para realizarla sin las consultas HQL, pero implican cargar la información de la base de datos a nuestro programa.

Estas consultas son más adecuadas para procesamiento de grandes volúmenes de información sin tener que recurrir a cargar la información a nuestro programa para tener que procesarla.

Consideraciones adicionales:

- **Parámetros:** En todos los ejemplos, se pueden pasar parámetros a las consultas utilizando el método **`setParameter()`** para hacer las consultas más dinámicas.
- **Uso de `where`:** La cláusula **`where`** es opcional en las consultas de actualización y borrado, pero si se omite, la operación afectará a todas las filas de la tabla.
- **executeUpdate():** Todas estas consultas se ejecutan mediante el método **`executeUpdate()`**, que retorna el número de filas afectadas por la operación.

CLAVES Y CONSEJOS

- Estas consultas son especialmente útiles para procesar grandes volúmenes de información sin necesidad de cargar todos los datos en la memoria de nuestro programa.
- Sin embargo, para operaciones más específicas y manipulación de objetos, Hibernate proporciona herramientas que permiten manipular los objetos directamente, aunque esto puede implicar cargar la información de la base de datos en nuestro programa.

HQL (Hibernate Query Language), veamos algunos ejemplos más desarrollados:

- Inserción:
  - Supongamos que tenemos una clase `Producto` y queremos insertar un nuevo producto en la base de datos utilizando una subconsulta para obtener los valores necesarios.
  - `String subconsulta = "SELECT nombre, precio FROM OtroObjeto WHERE condicion = :condicion";`
  - `Query query = session.createQuery("INSERT INTO Producto (nombre, precio) " + subconsulta);`
  - `query.setParameter("condicion", valorCondicion);`
  - `int filasAfectadas = query.executeUpdate();`
  - En este ejemplo, estamos insertando un nuevo producto en la tabla `Producto` utilizando los valores obtenidos de una subconsulta en la tabla `OtroObjeto`.
- Actualización:
  - Supongamos que queremos actualizar el precio de todos los productos que tengan una cierta condición.
  - `String hqlUpdate = "UPDATE Producto SET precio = :nuevoPrecio WHERE condicion = :condicion";`
  - `Query query = session.createQuery(hqlUpdate);`
  - `query.setParameter("nuevoPrecio", nuevoPrecio);`
  - `query.setParameter("condicion", valorCondicion);`
  - `int filasAfectadas = query.executeUpdate();`

Aquí, estamos actualizando el precio de los productos que cumplen con cierta condición especificada.

» Borrado:

- Supongamos que queremos borrar todos los productos que estén agotados.
- `String hqlDelete = "DELETE FROM Producto WHERE estado = :agotado";`
- `Query query = session.createQuery(hqlDelete);`
- `query.setParameter("agotado", Estado.AGOTADO);`
- `int filasAfectadas = query.executeUpdate();`
- Este ejemplo muestra cómo borrar todos los productos que tengan un estado específico (en este caso, agotado).

UAX FP