Unidad 3: Herramientas de mapeo objeto relacional (ORM).

Índice de contenidos

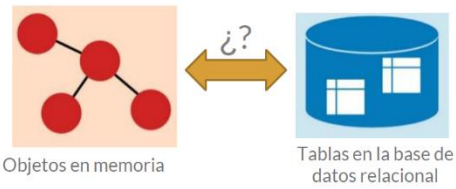
1 Persistencia de datos.

La persistencia de datos se refiere a la **capacidad de almacenar y recuperar datos a lo largo del tiempo**. Hay **distintas formas** de lograr la persistencia de datos, y la elección dependerá en gran medida de los requisitos específicos de la aplicación. Algunas de las opciones **más comunes** son:

Tipo	Ventajas	Desventajas
Sistemas de archivos	• Simple. • Adecuado para datos no estructurados o pequeñas cantidades de datos.	• Menos eficiente para operaciones complejas o grandes conjuntos de datos.
***SGBD* Relacionales**	• Estructura organizada. • Soporte para consultas complejas. • Integridad de datos.	• Puede ser más rígido para modelos de datos complejos.
***SGBD* NoSQL**	• Flexibilidad para manejar datos no estructurados. • Escalabilidad horizontal.	• Menos soporte para consultas complejas en comparación con las *BD* relacionales.
***SGBD* Orientados a Objetos**	• Alineación natural con el modelo de objetos. • Eliminan la necesidad de mapeo objeto-relacional.	• Complejidad de las consultas. • Menor madurez de mercado y comunidad. • Posibles problemas de rendimiento en grandes conjuntos de datos. • Dificultades de integración con herramientas existentes.

Las **desventajas de los sistemas de archivos, frente a los *SGBD***, para realizar la persistencia no invitan a su uso. **En esta unidad se va a estudiar la persistencia de datos en *SGBD* relacionales haciendo uso de un *ORM* (Object-Relational Mapping)**, en próximas unidades se verán el resto de opciones (NoSQL y Orientados a Objetos).

Realizar la **persistencia de datos en un *SGBD* relacional conlleva un desafío cuando la aplicación utiliza un enfoque basado en objetos**, mientras que el *SGBD* relacional emplea tablas para almacenar los datos. Esto **requiere una conversión bidireccional entre objetos y registros de tablas**. La técnica encargada de gestionar esta traducción se denomina **mapeo objeto-relacional**.

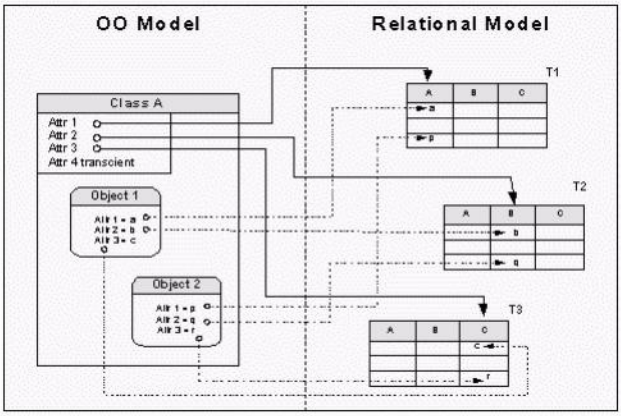


1.1 Desfase objeto-relacional.

El **desfase objeto-relacional** (también llamado impedancia objeto-relacional) **surge cuando una aplicación desarrollada con un lenguaje de programación orientado a objetos interactúa con una *BD* relacional**. Esta situación es común debido a la amplia adopción tanto de los lenguajes orientados a objetos como de las *BD* relacionales. Como consecuencia, el desarrollador debe dominar dos lenguajes distintos: el de programación y el específico para la gestión de la *BD*.

En el contexto del desfase objeto-relacional, considerar una aplicación desarrollada en *Java*. Por ejemplo, se podría tener una clase como la siguiente:

```java
public class Personaje {
    private int id;
    private String nombre;
    private String descripcion;
    private int vida;
    private int ataque;
    public Personaje(...) {
        ...
    }
    // getters y setters
}
```



En contraste, en la *BD* relacional, se tendría una tabla con campos que deben corresponderse con los atributos definidos en la clase:

```sql
CREATE TABLE personajes (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(50) NOT NULL,
  descripcion VARCHAR(50),
```

```
    vida INT DEFAULT 10,
    ataque INT DEFAULT 10
);
```

Dado que las clases y las tablas son estructuras conceptualmente diferentes, es necesario realizar un mapeo manual. Esto implica asociar los atributos de la clase con los campos de la tabla, utilizando métodos como los *getters* y *setters*.

Por ejemplo:

- Al escribir un objeto en la *BD*, el programador debe descomponer el objeto y construir manualmente la sentencia *SQL* necesaria para realizar la inserción, modificación o eliminación.
- Al leer datos de la *BD*, el programador debe reconstruir un objeto a partir de los valores obtenidos de la tabla, asignando cada valor a su atributo correspondiente.

Este proceso recae directamente sobre el programador, lo que supone un esfuerzo importante, especialmente en aplicaciones con muchas clases y tablas.

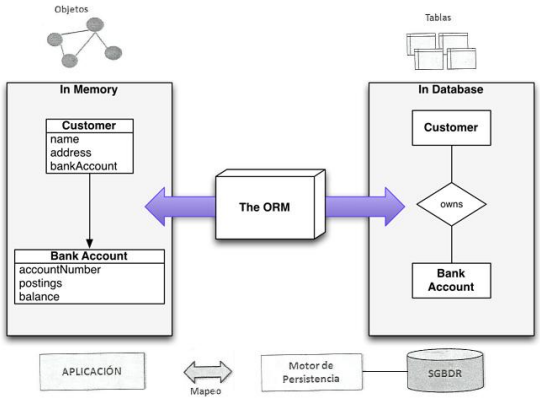
1.2 Técnicas de persistencia de objetos en *BD* relacionales.

Técnica	Descripción	Uso	Ventajas	Desventajas
***JDBC* (Java Database Connectivity)**	*API* de *Java* que permite conectarse a una *BD* relacional, enviar consultas y actualizar datos.	Requiere escribir sentencias *SQL* manualmente y gestionar conexión/recursos explícitamente.	– Control total sobre las consultas y la conexión.	– Requiere más código. – Propenso a errores. – Menos productivo.
***JPA* (Jakarta Persistence API)**	Especificación *Java* para mapeo objeto-relacional; implementaciones como *Hibernate* o *EclipseLink*.	Uso de anotaciones en clases *Java* para mapearlas a tablas de *BD*.	– Consultas *JPQL* (Java Persistence Query Language). – Facilita el desarrollo.	– Implementaciones pueden tener comportamientos específicos. – Se necesita aprender *JPQL*.
Frameworks *ORM* (Object-Relational Mapping)	Proporcionan una capa de abstracción para simplificar el acceso y manipulación de datos. Ej.: *Hibernate* o *MyBatis*.	Configurar el framework, definir mapeos objeto-relacional y usar la *API* específica.	– Reduce el código repetitivo. – Manejo automático de conexiones y transacciones.	– Mayor curva de aprendizaje. – Posible pérdida de control sobre las consultas.

1.3 Mapeo objeto-relacional.

El mapeo objeto-relacional (*ORM*) es una **técnica de programación utilizada para simplificar la interacción entre sistemas orientados a objetos y *BD* relacionales**. Permite convertir los datos de los objetos del programa en registros de las tablas de la *BD* y viceversa, sin necesidad de realizar operaciones directas en la *BD*.

En esencia, el ***ORM* actúa como una capa de abstracción que enlaza los objetos del código con las tablas de la *BD***. Esto elimina la necesidad de escribir sentencias *SQL* manualmente, ya que las **operaciones sobre los datos se realizan mediante objetos y métodos en el lenguaje de programación**.



El **objetivo principal** del *ORM* es **simplificar el desarrollo y mejorar el mantenimiento del código al reducir la cantidad de código repetitivo relacionado con la interacción con la *BD***. Al utilizar un *ORM*, los desarrolladores pueden trabajar directamente con clases y objetos, sin preocuparse por los detalles específicos de la *BD*.

Esta técnica es especialmente útil en entornos que emplean el paradigma de programación orientada a objetos (*POO*), ya que integra de manera fluida la lógica del programa con el almacenamiento de datos. No obstante, su uso excesivo puede generar problemas de rendimiento debido a la sobrecarga que implican las abstracciones adicionales.

Cuando se trabaja directamente con *JDBC*, es necesario descomponer el objeto en sus atributos para construir la sentencia `INSERT` correspondiente. Por ejemplo:

```
String sentenciaSql = "INSERT INTO personajes(nombre, descripcion, vida, ataque) VALUES (?, ?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sentenciaSql);
sentencia.setString(1, personaje.getNombre());
sentencia.setString(2, personaje.getDescripcion());
```

```
        sentencia.setInt(3, personaje.getVida());
        sentencia.setInt(4, personaje.getAtaque());
        sentencia.executeUpdate();
        sentencia.close();
```

Sin embargo, al trabajar con *JPA* directamente, la operación de persistencia se simplifica significativamente. Se puede trabajar directamente con el objeto *Java*, ya que *JPA* se encarga de realizar el mapeo entre la clase y la tabla de la *BD* utilizando las anotaciones definidas en la clase. Por ejemplo:

```
@Entity
@Table(name="personajes")
public class Personaje {
    @Id // Marca el campo como la clave de la tabla
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private int id;
    @Column(name="nombre")
    private String nombre;
    @Column(name="descripcion")
    private String descripcion;
    @Column(name="vida")
    private int vida;
    @Column(name="ataque")
    private int ataque;

    public Personaje(String nombre, String descripcion, int vida, int ataque) {
        this.nombre = nombre;
        this.descripcion = descripcion;
        this.vida = vida;
        this.ataque = ataque;
    }

    // getters y setters
}
```

El **mapeo Objeto-Relacional**, ***ORM*** en inglés, es una técnica utilizada para **convertir clases** (y sus relaciones) de un sistema que utiliza un **lenguaje de programación orientado a objetos** a el modelo de una ***BD* relacional**.

¿Cómo hace un *ORM* este mapeo? → Mediante una serie de "indicaciones" que se realizan utilizando un concepto llamado "**annotations**".

Con esta configuración, basta con establecer una conexión a la *BD* y enviar el objeto directamente. La operación de persistencia se realiza con *JPA* mediante el `EntityManager`:

```
// Crear la instancia del EntityManagerFactory
EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUnidadDePersistencia");

try (EntityManager em = emf.createEntityManager()) {
    em.getTransaction().begin(); // Iniciar una transacción
    // Crear un objeto Personaje y establecer sus atributos
    Personaje personaje = new Personaje("Gandalf", "Mago blanco", 100, 50);
    em.persist(personaje); // Persistir el objeto en la BD
    em.getTransaction().commit(); // Confirmar la transacción
} catch (Exception e) {
    System.out.println("Error al persistir el objeto: " + e.getMessage());
}
```

Este enfoque abstrae la lógica de interacción con la *BD* y permite concentrarse en la lógica del negocio. *JPA* no solo reduce el código necesario para persistir datos, sino que también facilita el desarrollo al manejar automáticamente el mapeo entre clases y tablas, las relaciones, y las transacciones de manera eficiente.

1.4 Herramientas *ORM*.

Dentro del concepto de mapeo objeto-relacional, se utilizan **herramientas y frameworks especializados** que facilitan la implementación de esta técnica. Algunos ejemplos destacados son ***Hibernate* para *Java***, *Entity Framework* para *.NET*, *Django ORM* para *Python* y *Sequelize* para *JavaScript/Node.js*.

Estas herramientas **suelen ofrecer funcionalidades adicionales, como la gestión automática de esquemas de *BD*, la optimización de consultas y la manipulación eficiente de transacciones**. Además, permiten establecer relaciones entre objetos en el código, reflejando de manera más natural las relaciones entre tablas en la *BD*.

1.5 Ventajas e inconvenientes de los *ORM*.

Los *ORMs* son herramientas poderosas, pero como cualquier tecnología, presentan tanto ventajas como inconvenientes:

- **Ventajas**:
  - ✓ **Reducción del tiempo de desarrollo de software**.
  - ✓ **Abstracción de la *BD***: trabajan con objetos en lugar de preocuparse por la sintaxis específica de *SQL*.
  - ✓ **Productividad**: reduce la cantidad de código que se necesita escribir para interactuar con la *BD*.
  - ✓ **Portabilidad de *BD***: facilita el cambio de *SGBD* sin tener que reescribir el código.
  - ✓ **Mapeo objeto-relacional**: permite modelar la *BD* en términos de objetos y relaciones.
  - ✓ **Gestión de transacciones**: simplifica la gestión de transacciones.
- **Desventajas:**
  - ✓ **Rendimiento**: las consultas generadas por el *ORM* pueden no ser tan eficientes.
  - ✓ **Curva de aprendizaje**: hay que familiarizarse con las convenciones y configuraciones específicas.
  - ✓ **Complejidad**: en aplicaciones complejas, la complejidad de las consultas generadas por el *ORM* puede aumentar. En tales casos, algunos desarrolladores prefieren un control más directo sobre las consultas.
  - ✓ **Personalización y control**: en ciertos casos, puede ser necesario un control más fino sobre las consultas o la estructura de la *BD*, y los *ORMs* pueden limitar este nivel de personalización.
  - ✓ **Posible abstracción excesiva**: *ORM* puede ocultar detalles importantes de la *BD*, lo que puede llevar a decisiones de diseño que no sean óptimas.

En general, los *ORMs* pueden aumentar la productividad y facilitar el desarrollo de aplicaciones. Sin embargo, **es esencial comprender sus limitaciones y ajustar su uso según las necesidades específicas del proyecto**.

Situaciones en las que puede ser beneficioso utilizar un *ORM*:

- ✓ **Desarrollo rápido y productividad**: si se está desarrollando una aplicación de forma rápida y se desea minimizar la cantidad de código necesario para interactuar con la *BD*.
- ✓ **Modelado orientado a objetos**: cuando se está trabajando en un entorno orientado a objetos y se desea modelar la *BD* de manera más natural en términos de objetos y relaciones.
- ✓ **Portabilidad de *BD***: si se necesita la capacidad de cambiar fácilmente entre proveedores de *BD* sin tener que reescribir grandes partes del código.
- ✓ **Aplicaciones de tamaño mediano a grande**: en aplicaciones grandes y complejas, donde la gestión manual de consultas *SQL* puede volverse complicada, un *ORM* puede simplificar el desarrollo al proporcionar una capa de abstracción.
- ✓ **Evitar *SQL Injection***: los *ORMs* suelen ofrecer protección contra ataques de inyección *SQL*, ya que utilizan consultas parametrizadas y gestionan la interacción con la *BD* de manera segura.

Situaciones en las que podría ser preferible evitar el uso de un *ORM*:

- ✓ **Requisitos de rendimiento críticos**: en situaciones en las que el rendimiento es absolutamente crítico y cada milisegundo cuenta, algunas operaciones manuales de *SQL* pueden ser más eficientes que las generadas automáticamente por un *ORM*.
- ✓ **Necesidad de control preciso sobre consultas**: si se tienen consultas *SQL* complejas y se requiere un control más detallado sobre cómo se ejecutan, puede ser más adecuado escribir las consultas manualmente.
- ✓ **Proyectos pequeños y simples**: en proyectos pequeños donde la complejidad de un *ORM* puede ser excesiva, y las operaciones en la *BD* son simples, podría ser más fácil y directo interactuar directamente con *SQL*.

2 *JPA*.

2.1 ¿Qué es *JPA*?

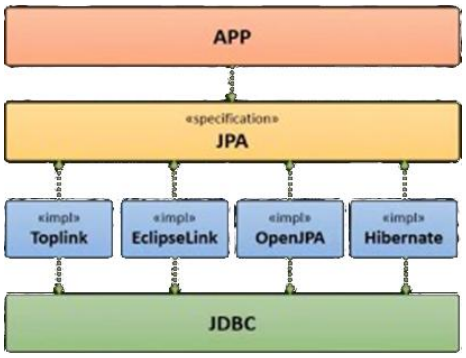
JPA es un ORM (Object Relational Mapping) que tiene como objetivo lograr la persistencia de datos entre una aplicación desarrollada en *Java* y una *BD*.

JPA (*Java*/***Jakarta Persistence API***) es una **especificación de *Java*** (la *JSR* 338) **que describe la gestión de datos y la relación entre objetos *Java* y las tablas en *BD* relacionales**. **Define cómo deben utilizarse las anotaciones para persistir objetos, cómo deben buscarse, cuál es su ciclo de vida, etc.** *JPA* traduce el modelado de las clases *Java* al modelado relacional en una *BD*, permitiendo a los programadores decidir qué clases u objetos desean persistir.

Una especificación es un documento en el que se define como se debe gestionar una funcionalidad.

Al tratarse de un documento, *JPA* **no implementa nada** directamente; para trabajar con ella **es necesario utilizar un framework que implemente la especificación**.

La **pregunta clave suele ser si se debe usar directamente la especificación u optar por un framework que la implemente**. Muchos desarrolladores prefieren usar el framework directamente debido a las capacidades adicionales que ofrecen, las cuales no están soportadas por *JPA* como estándar y especificación. Sin embargo, hay desarrolladores que defienden el uso exclusivo de *JPA*, argumentando que, aunque un framework puede aportar algunas ventajas adicionales, estas no son críticas en la mayoría de las situaciones y las ventajas de usar directamente la especificación superan los beneficios de las mejoras adicionales de los frameworks.



JPA en sí misma es solo una especificación y no proporciona la implementación concreta de cómo interactuar con la BD. El proveedor de persistencia es quien implementa las interfaces y especificaciones definidas por JPA. Algunos de los proveedores de persistencia más usados para JPA son Hibernate, EclipseLink, Toplink, MyBatis y OpenJPA.

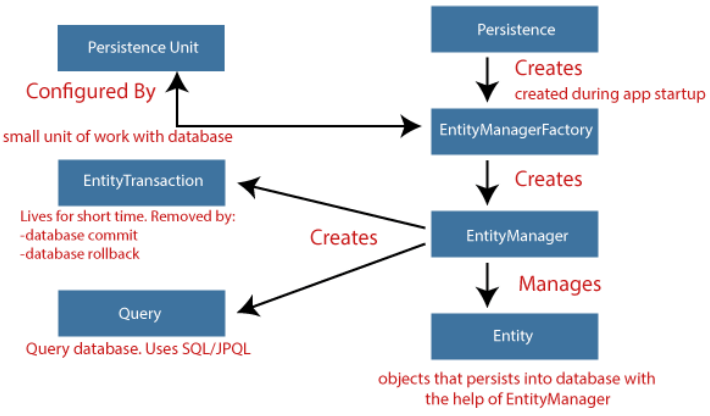
Ventajas de usar la especificación:

- ✓ La especificación está **muy trabajada y** es **ampliamente conocida entre los desarrolladores**. El uso concreto de un framework no lo está tanto.
- ✓ La especificación **permite cambiar de implementación de forma transparente en caso de necesidad**. Simplifica el proceso de trasladar aplicaciones entre diferentes proveedores de persistencia.
- ✓ La especificación **es más atemporal debido a su carácter documental**, ya que **todos los desarrolladores se ajustan a ella**. Esto hace que los conocimientos adquiridos sean más duraderos en el tiempo y más sólidos.
- ✓ **Es habitual que surjan nuevos frameworks o especificaciones que complementen y amplíen *JPA***.

2.2 Arquitectura.

La arquitectura de *JPA* está **diseñada para gestionar entidades y las relaciones que hay entre ellas**. Sus **principales componentes** son (paquete `jakarta.persistence`):

- ✓ **`Persistence`**: **clase con métodos estáticos que permiten obtener instancias de `EntityManagerFactory`**.
- ✓ **`EntityManagerFactory`**: esta interfaz actúa como una factoría de `EntityManager`. **Se encarga de crear y gestionar múltiples instancias de `EntityManager`**.
- ✓ **`EntityManager`**: es una interfaz que **gestiona las operaciones de persistencia de las entidades**. Funciona también como una factoría de `Query`.
- ✓ **`Query`**: es una interfaz que permite **obtener la lista de objetos que cumplen con un criterio**.
- ✓ **`EntityTransaction`**: es una interfaz que **agrupa las operaciones realizadas sobre un `EntityManager` en una única transacción** de *BD*.
- ✓ **`Entity`**: representan los **objetos persistidos en *BD*** como registros de una tabla. Cada entidad se corresponde con una tabla en la *BD*.



Más información https://jakarta.ee/specifications/persistence/ y https://jakarta.ee/specifications/persistence/3.1/jakarta-persistence-spec-3.1.html

2.3 Instalación.

Para utilizar *JPA* en un proyecto, es necesario **agregar las dependencias correspondientes en el archivo de configuración del sistema de gestión de dependencias**, ya sea *Maven* o *Gradle*:

- ***Maven***: agregar las dependencias en el archivo **`pom.xml`** dentro de la **sección `<dependencies>`**.

```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
    <version>3.5.1</version>
</dependency>
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.5.2.Final</version>
</dependency>
```

JPA busca traducir el modelado de las clases *Java* a un modelado relacional en una *BD*, posibilitando al programador elegir qué clases u objetos quiere persistir.

- ***Gradle***: agregar las dependencias en el archivo **`build.gradle`** dentro de la **sección `dependencies`**.

```
implementation 'org.mariadb.jdbc:mariadb-java-client:3.5.1'
implementation 'org.hibernate:hibernate-core:6.5.2.Final'
```

Es importante incluir las dependencias *JDBC* correspondientes al *SGBD* que se utilizará en el proyecto.

Esta configuración permitirá integrar *JPA* en el proyecto y habilitar la persistencia de datos a través de un proveedor de persistencia como *Hibernate*.

2.4 Configuración de la unidad de persistencia.

Hay que **asegurarse de que el controlador *JDBC* del *SGBD* que se vaya a utilizar se haya agregado al proyecto**.

La configuración de *JPA* se realiza en el **archivo de configuración `persistence.xml`** y la **carpeta** donde se debe colocar es **`src/main/resources/META-INF`**.

Ejemplo: configurar *JPA* con *MariaDB* como *SGBD*.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
        https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
        <!-- Nombre de la unidad de persistencia -->
        <persistence-unit name="nombre_unidad_persistencia" transaction-type="RESOURCE_LOCAL">
        <description>Ejemplo XXXXXXXXX básico JPA con la implementación de Hibernate</description>
        <!-- Se indica el "provider" que es la implementación de JPA que se está usando. -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- Se definen las clases que representan "entidades". Por cada clase se debe utilizar la etiqueta
            <class> cuyo contenido debe incluir el paquete y el nombre de la clase: -->
        <class>paquete.ClasePersistente1</class>
        ...
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <!-- Propiedades de configuración de la conexión al SGBD MariaDB y a la BD especificada -->
            <property name="jakarta.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver"/>
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost:3306/basededatos"/>
            <property name="jakarta.persistence.jdbc.user" value="usuario"/>
            <property name="jakarta.persistence.jdbc.password" value="contraseña"/>
            <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
            <!– No hace nada con el esquema de la BD -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>
            <!-- Para ver las consultas SQL (OPCIONAL) -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Cuando se establece en **`true`, solo se escanearán las clases enumeradas en el archivo de configuración** en busca de clases persistentes (anotadas con `@Entity`). El **valor por defecto si no se indica nada es `false`** que indica que *JPA* **debe incluir todas las clases que se encuentren en el CLASSPATH**.

Se pueden agregar varias unidades de persistencia dentro de un mismo archivo `persistence.xml`

La propiedad **`jakarta.persistence.schema-generation.database.action` permite controlar la generación y actualización automática del esquema de la *BD* basándose en las entidades mapeadas**. **Valores** que puede tomar:

- ✓ **`none`**: no se realiza ninguna acción en la *BD*. Es el valor predeterminado si no se especifica explícitamente.
- ✓ **`create`**: crea las tablas y estructuras del esquema en la *BD*. Si las tablas ya existen, puede dar errores

dependiendo del proveedor.

> En la versión 3.2 de *JPA* se incorpora `validate`.

- ✓ **`drop`**: elimina la *BD* sin crear una nueva.
- ✓ **`drop-and-create`**: elimina el esquema de la *BD* y después lo crea nuevamente.

Opciones adicionales como `update` o `validate` **no forman parte del estándar**, pero **algunos proveedores como *Hibernate* las soportan** de forma específica:

- ✓ **`update`**: actualiza la *BD* sin eliminar datos.
- ✓ **`validate`**: valida que el esquema existente coincide con las entidades mapeadas. No realiza cambios.

Estos valores **deben configurarse adecuadamente según el entorno**:

- ✓ En **desarrollo**, **`create`**, **`drop`** o **`drop-and-create`** suelen ser útiles para pruebas rápidas.
- ✓ En **producción**, es común usar **`none`** o **`validate`** para evitar perder datos.

2.5 Crear objeto `EntityManager` para interactuar con la *BD*.

Para interactuar con la *BD* (realizar consultas o insertar, actualizar y eliminar datos), **es necesario utilizar una instancia del gestor de entidades *JPA***.

El **nombre especificado en `"nombre_unidad_persistencia"` debe coincidir con el definido en el archivo `persistence.xml`**. A continuación, se muestra cómo configurar el gestor de entidades:

```
EntityManagerFactory entityManagerFactory =
                            Persistence.createEntityManagerFactory("nombre_unidad_persistencia");
EntityManager entityManager = entityManagerFactory.createEntityManager();
// Operaciones con el entityManager
// Cerrar el EntityManager y el EntityManagerFactory al finalizar la aplicación
entityManager.close();
entityManagerFactory.close();
```

Recordar **cerrar siempre el `EntityManager` y el `EntityManagerFactory` para liberar recursos al finalizar la aplicación**.

Ejemplo: integración de *JPA* en un proyecto.

1. **Agregar las dependencias** en el archivo **`pom.xml`**: incluir las dependencias necesarias, como el driver *JDBC* del *SGBD* (por ejemplo, *MariaDB*) y la implementación de *JPA* (por ejemplo, *Hibernate*).

```
<dependencies>
    <dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client</artifactId>
        <version>3.5.1</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>2.0.16</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.5.2.Final</version>
    </dependency>
</dependencies>
```

> Cada `EntityManager` está asociado con un contexto y puede realizar operaciones de persistencia, consulta y gestión de entidades en ese contexto.

2. **Crear el archivo de configuración `persistence.xml`** en **`src/main/resources/META-INF`**.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
        https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
    <persistence-unit name="unidad_persistencia" transaction-type="RESOURCE_LOCAL">
        <description>Ejemplo básico JPA con la implementación de Hibernate</description>
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <!-- Configuración de acceso a la BD en MariaDB -->
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost:3306/basededatos"/>
            <property name="jakarta.persistence.jdbc.user" value="usuario"/>
            <property name="jakarta.persistence.jdbc.password" value="contraseña"/>
            <!-- Configuración del esquema de la BD - No hace nada con el esquema -->
```

```
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>
            <!-- Mostrar sentencias SQL en consola -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

> ***JPA* no crea automáticamente la *BD***, solo las tablas dentro de una *BD* ya existente. La *BD* se debe crear manualmente o usar `?createDatabaseIfNotExist=true` en la *URL JDBC*.
>
> Si se usa `?createDatabaseIfNotExist=true` el usuario configurado en la conexión debe tener permisos para crear *BD*.

3. **Programa básico *Java* que crea el gestor de entidades *JPA***.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        // "unidad_persistencia" debe coincidir con el nombre en el archivo persistence.xml
        try (EntityManagerFactory entityManagerFactory =
                Persistence.createEntityManagerFactory("unidad_persistencia");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            System.out.println("\n\n**** Gestor de entidades JPA creado correctamente. ****\n\n");

            // Realizar operaciones con el entityManager (se verán en próximos apartados)
        }
    }
}
```

3 Mapeo de entidades, atributos y relaciones.

JPA* utiliza una serie de mapeos que deben aplicarse a los elementos de una clase para establecer la relación entre la clase y la *BD. Estos mapeos **se representan mediante anotaciones (@)**.

Para realizar el mapeo de entidades y relaciones en *JPA*, **es necesario crear clases *Java* que representen las entidades y mapearlas a tablas en la *BD* utilizando anotaciones** *JPA*.

En el **caso de las entidades**:

- ✓ La **clase debe estar anotada** para indicar con qué tabla se mapeará.
- ✓ **Cada atributo** (o los métodos *getter*) **también debe estar anotado** para especificar los campos correspondientes en la tabla.

JPA* permite definir las anotaciones como `@Id`, `@Column`, etc., tanto en los atributos como en los métodos *getter, pero **nunca en los métodos *setter***.

La **decisión sobre dónde anotarlas la determina la anotación `@Id`**. El **lugar donde esta se aplique (ya sea en los atributos o en los *getter*) establece la estrategia que se seguirá para el resto de las anotaciones de la clase** (esta elección debe ser consistente dentro de la clase para evitar errores durante la ejecución). Por lo tanto:

- ✓ Si `@Id` se coloca sobre los atributos, todas las demás anotaciones deben colocarse también sobre los atributos.
- ✓ Si `@Id` se coloca sobre los *getter*, las demás anotaciones deben colocarse en los *getter*.

Al mapear una entidad en *JPA*, todos sus atributos son considerados persistentes de forma predeterminada. Esto significa que, salvo que se indique lo contrario, cada atributo se almacenará como una columna en la tabla correspondiente de la *BD*.

3.1 Entidades.

Las **anotaciones** disponibles para **mapear una clase *Java* a una tabla** en la *BD* son las siguientes:

- ✓ **`@Entity`**: se utiliza para **marcar una clase como una entidad persistente**. Esto significa que **la clase se mapeará a una tabla** en la *BD* y cada instancia de esta clase representará una fila en dicha tabla. Esta anotación es **obligatoria** y se coloca al inicio de la definición de una clase.

```
@Entity                          import jakarta.persistence.Entity;
public class MiEntidad {
    // Atributos y métodos
}
```

> **Una entidad en *JPA* debe ser una clase anotada con `@Entity`, tener un campo identificador único anotado con `@Id` y un constructor sin parámetros.**

- ✓ **`@Table(name = "nombre_tabla"[, catalog = "nombre_bd"[, schema="nombre_esquema"]])`**: es **opcional y permite especificar la tabla de la *BD* a la que se asignará la entidad**, útil si el nombre de la tabla es diferente al de la clase.
  - **`name`**: define el **nombre de la tabla en la *BD***.
  - `catalog` (opcional): especifica el catálogo al que pertenece la tabla.
  - `schema` (opcional): indica el esquema donde se encuentra la tabla.

  Estos últimos parámetros (`catalog` y `schema`) generalmente no son necesarios, ya que la configuración suele estar definida en el archivo `persistence.xml`.

  Además, permite definir **restricciones de unicidad** a través de **`uniqueConstraints`** y también ofrece la opción de definir **índices adicionales** con **`indexes`** para mejorar el rendimiento de las consultas.

  Si no se utiliza `@Table`, *JPA* asignará como nombre de la tabla el nombre de la clase.

```
@Entity
@Table(name = "nombre_tabla", catalog = "nombre_bd", schema = "nombre_esquema")
public class MiEntidad {
    // Atributos y métodos
}
```

```
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
```

  Con estas anotaciones se puede configurar la relación básica entre una clase y su tabla en la *BD*.

- ✓ **`@MappedSuperclass`**: indica que una **clase no será mapeada directamente a una tabla en la *BD*, pero sus atributos y métodos serán heredados y utilizados por las entidades que la extiendan**.

  Esta anotación se utiliza para definir una clase base común que contiene atributos y métodos compartidos por varias entidades, sin que la propia clase sea persistente.

  Características principales:
  - La clase anotada con `@MappedSuperclass` no se convierte en una tabla en la *BD*.
  - Las **clases que heredan de ella sí serán entidades persistentes y sus atributos incluirán los definidos en la clase base**.
  - Es útil para evitar duplicación de código en entidades con atributos comunes, como `id`, `creadoPor`, `actualizadoPor`, etc.

```
import jakarta.persistence.Id;
import jakarta.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class EntidadBase {
    @Id
    private Long id;
    private String creadoPor;

    // Métodos comunes
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getCreadoPor() { return creadoPor; }
    public void setCreadoPor(String creadoPor) { this.creadoPor = creadoPor; }
}

import jakarta.persistence.Entity;

@Entity
public class Producto extends EntidadBase {
    private String nombre;
    private Double precio;

    // Métodos específicos de la clase Producto
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public Double getPrecio() { return precio; }
    public void setPrecio(Double precio) { this.precio = precio; }
}
```

  La clase anotada con `@MappedSuperclass` no necesariamente tiene que ser abstracta, pero suele ser una buena práctica marcarla como abstracta cuando no tiene sentido instanciarla directamente.

- ✓ **`@Embeddable`**: se utiliza para **marcar una clase cuyos objetos deben ser embebidos** (incrustados) **como componentes en la entidad que la contiene**.

  Esto implica que los **atributos de la clase embebida se incorporarán directamente en la tabla de la entidad que la contiene**, como si fueran parte de esa entidad. Es decir, no se crea una tabla separada para la clase embebida, sino que sus campos se agregan a la tabla de la entidad que la usa.

  En el siguiente ejemplo, la clase `Direccion` está marcada con `@Embeddable`, lo que indica que sus atributos

(`calle`, `ciudad`, `codigoPostal`) se agregarán a la tabla de la entidad `Persona`.

```
@Embeddable
public class Direccion {
    private String calle;
    private String ciudad;
    private String codigoPostal;
    // Getters y setters
}
```

```
import jakarta.persistence.Embeddable;
```

```
@Entity
public class Persona {
    @Id
    private Long id;
    private String nombre;
    @Embedded
    private Direccion direccion;

    // Getters y setters
}
```

```
import jakarta.persistence.Embedded;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
```

```
// Los atributos de Direccion serán
// parte de la tabla Persona
```

No es posible combinar las anotaciones `@Entity`, `@MappedSuperclass` y `@Embeddable` en una misma clase, ya que tienen propósitos diferentes. Sin embargo, sí **es posible usar estas anotaciones de manera separada en una jerarquía de clases**, es decir, una clase puede ser anotada con `@Entity`, otra con `@MappedSuperclass`, y otra con `@Embeddable`.

En cuanto a la **anotación `@Table`, esta solo se puede combinar con `@Entity` en una misma clase**. No es compatible con `@MappedSuperclass` ni con `@Embeddable`.

3.2 Atributos.

Las **anotaciones** que se utilizan en *JPA* para **anotar los atributos que deben ser mapeados a los campos de la tabla** correspondiente son las siguientes:

- ✓ **`@Id`**: se utiliza para **marcar un atributo como la clave primaria de la entidad**. **Cada clase de entidad debe declarar un atributo anotado con `@Id`, o heredar uno de una `@MappedSuperclass`**. En una entidad subclase, el atributo identificador se hereda de la entidad padre, por lo que no puede declarar su propio atributo `@Id`.
  - **Generados y asignados por *JPA***: **`@GeneratedValue(strategy = GenerationType.ESTRATEGIA)`**. Existen varias **estrategias de asignación**:
    - **`GenerationType.AUTO`**: selecciona automáticamente la **mejor estrategia en función del *SGBD***.
    - **`GenerationType.IDENTITY`**: utiliza una **columna especial,** que puede ser **autonumérica** en ciertos *SGBD* (como *MariaDB*), **o seguir una política específica que garantice la unicidad de la clave**.

```
@Entity
public class Orden {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String descripcion;
}
```

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

    - **`GenerationType.SEQUENCE`**: usa una **secuencia *SQL* para generar los valores** de todas las claves.
    - **`GenerationType.TABLE`**: usa una **tabla adicional en la *BD***. Cada **fila representa un tipo de entidad distinto y almacena el siguiente valor disponible para la clave**.

Requieren configuración adicional.

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "producto_gen")
@TableGenerator(name = "producto_gen", table = "id_generator")
private Long id;
```

  - **Asignados por la aplicación** (claves naturales): en este caso, el atributo identificador no se anota con `@GeneratedValue`, **sino que solo se anota con `@Id`**. Es **responsabilidad de la aplicación asignar un valor al atributo identificador**.

```
@Entity
public class Producto {
    @Id
```

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
```

```
        private String codigo; // La aplicación debe asignar este valor
        private String nombre;
    }
```

Las claves naturales, que consisten en múltiples columnas (**claves compuestas**), se pueden representar con **varios campos anotados con `@Id`**. Sin embargo, una mejor opción es usar **`@IdClass`** (creando una clase independiente con los campos que coincidan con los identificadores de la entidad) o **`@EmbeddedId`**. Ejemplo:

```
@Embeddable
public class ProductoId implements Serializable {
    private String codigo;
    private String sucursal;
    // Constructor, getters y setters. Métodos equals() y hashCode() se deben implementar
}
```

Igual para las dos formas.

- Con **`@EmbeddedId`**:

```
@Entity
@Table(name = "productos")
public class Producto {
    @EmbeddedId
    private ProductoId id;
    // Resto de atributos
    private String nombre;
    // Constructor sin argumentos, getters y setters.
}
```

Una **clave primaria compuesta es una clave primaria conformada por más de un atributo en una entidad**. Aunque resulta **más cómodo trabajar con claves primarias simples**, hay situaciones en las que puede ser obligatorio trabajar con claves primarias compuestas, por ejemplo, en *BD* ya existentes que las tengan ya definidas.

- Con **`@IdClass`**:

```
@Entity
@Table(name = "productos")
@IdClass(ProductoId.class)
public class Producto {
    @Id
    private String codigo;
    @Id
    private String sucursal;
    // Resto de atributos
    private String nombre;
    // Constructor sin argumentos, getters y setters.
}
```

La interfaz `Serializable` es necesaria cuando se utilizan identificadores compuestos con `@IdClass` o `@EmbeddedId` en *JPA*. Esto se debe a que *JPA* requiere que los identificadores sean serializables para garantizar que puedan ser correctamente transferidos, almacenados o utilizados en entornos distribuidos o en cachés. Además, al implementar `Serializable`, es importante sobrescribir los métodos `equals()` y `hashCode()` para que *JPA* pueda comparar y gestionar adecuadamente los objetos del identificador compuesto.

- ✓ **`@Column`**: esta anotación se utiliza para **especificar detalles sobre la columna de la tabla a la que se asigna un atributo** de la entidad, permitiendo personalizar el mapeo. **Si no se utiliza, el nombre de la columna en la tabla será el mismo que el nombre del atributo en la clase**. Algunas **opciones comunes** que proporciona son las siguientes:
  - **`name`**: permite especificar el **nombre** de la columna en la tabla. Ej.: `@Column(name = "nombre_columna")`
  - **`nullable`**: indica si la columna **puede contener valores nulos**. Ej.: `@Column(nullable = false)`
  - **`length`**: define la **longitud máxima** de la columna (para los `String`), útil para campos de tipo cadena. Ej.: `@Column(length = 100)`
  - **`unique`**: especifica si los **valores** en la columna **deben ser únicos**. Ej.: `@Column(unique = true)`
  - **`updatable`** e **`insertable`**: controlan si la columna **debe incluirse en las operaciones de actualización o inserción**. Ej.: `@Column(updatable = false, insertable = false)`
  - **`columnDefinition`**: permite **especificar el tipo de datos *SQL* para la columna *SQL*** cuando se genera la tabla. Ej.: `@Column(columnDefinition = "TEXT")`
  - **`precision`** y **`scale`**: para campos con números decimales, **especifican la precisión y la escala de la columna** (`BigDecimal` → `DECIMAL(p,s)`).

```
@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nombre_producto", length = 50, nullable = false)
    private String nombre;

    @Column(unique = true, nullable = false)
    private String codigo;

    @Column(columnDefinition = "TEXT")
    private String descripcion;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal precio; // Dos decimales

    @Column(name = "fecha_creacion", updatable = false)
    private LocalDate fechaCreacion;

    @PrePersist
    private void prePersist() {
        this.fechaCreacion = LocalDate.now();
    }
}
```

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.PrePersist;
import java.math.BigDecimal;
import java.time.LocalDate;
```

La anotación `@PrePersist` se utiliza en *JPA* para definir un método que debe ejecutarse automáticamente antes de que una nueva entidad se inserte en la *BD*. Es útil para realizar inicializaciones o validaciones antes de guardar la entidad. Se tratará en el punto 5 de la unidad.

Si se necesita **garantizar que varios campos juntos sean únicos** (una combinación única de valores), se puede usar una clave única compuesta en *JPA*. Esto **se logra aplicando la anotación `@Table` con el atributo `uniqueConstraints`**.

Ejemplo:

```
@Entity
@Table(name = "usuarios",
    uniqueConstraints = {@UniqueConstraint(columnNames = {"email", "telefono"})})
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String email;

    @Column(nullable = false)
    private String telefono;

    private String nombre;

    // Getters y Setters
}
```

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import jakarta.persistence.UniqueConstraint;
```

- ✓ **`@Basic`**: se utiliza **para marcar un campo como persistente y proporciona opciones básicas de configuración**. Aunque **no es necesario utilizar `@Basic`**, ya que por defecto todos los campos son persistentes, esta anotación permite **expresar explícitamente la intención de persistencia**. También puede utilizarse para aplicar configuraciones adicionales. `@Basic(optional = false)` → significa que la propiedad no puede ser nula.

La **anotación `@Basic`** en *JPA* permite personalizar si un atributo puede ser `null` mediante **`optional`** (por defecto `true`) y su modo de carga con **`fetch`**, que es inmediata (`FetchType.EAGER`) por defecto o diferida (`FetchType.LAZY`).

Los atributos de una entidad que no se anotan por defecto hacen uso de `@Basic` y no es necesario ponerla.

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Basic(optional = false)
    private String nombre; // Este campo es obligatorio y no puede ser nulo

    @Basic
    private String correoElectronico; // Este campo puede ser nulo
}
```

- ✓ **`@Transient`**: se utiliza para **marcar un atributo como no persistente**, lo que significa que su valor no se almacenará en la *BD*. Este atributo será ignorado por *JPA* y solo existirá en la clase como parte de la lógica de la aplicación. Puede ser **útil para campos calculados o datos temporales** que no necesitan ser guardados en

la tabla correspondiente.

```
@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private double precioUnitario;
    @Transient
    private double precioTotal; // Este campo no será persistido en la BD
    public double calcularPrecioTotal(int cantidad) {
        return precioUnitario * cantidad;
    }
    // Métodos getter y setter
    ...
}
```

- ✓ **`@Embedded`**: se utiliza para **indicar que un atributo de una entidad debe ser embebido** (incrustado) **dentro de la entidad**. Esta anotación **se emplea junto con `@Embeddable`**, que se coloca en la clase que representa los componentes embebidos, permitiendo que esa clase forme parte de la entidad principal.

```
@Entity
public class Empleado {
    @Id
    private Long id;
    private String nombre;
    @Embedded
    private Direccion direccion; // La clase Direccion está incrustada dentro de Empleado
}

@Embeddable
public class Direccion {
    private String calle;
    private String ciudad;
    private String codigoPostal;
    // Getters y Setters
}
```

- ✓ **`@OrderBy`**: se utiliza **para especificar que los elementos de una colección o lista deben ordenarse según el criterio que se indique**. El valor por defecto es el orden ascendente (`asc`), pero también se puede definir el orden descendente (`desc`).

```
@Entity
public class Departamento {
    @Id
    private Long id;
    private String nombre;
    @OneToMany
    @OrderBy("nombre desc") // empleados se ordenará por nombre de manera descendente
    private List<Empleado> empleados;
}

@Entity
public class Empleado {
    @Id
    private Long id;
    private String nombre;
}
```

Si se desea ordenar los resultados de una consulta, puede hacerse mediante *JPQL* (Java Persistence Query Language), que permite especificar orden en la consulta directamente, no en la definición de la entidad.

- ✓ **`@Temporal`**: se utiliza para **mapear atributos de tipo `java.util.Date` o `java.util.Calendar` a una columna de una tabla que almacene una fecha**. Dependiendo del tipo de precisión que se quiera, se puede utilizar uno de los siguientes **valores de `TemporalType`**:
  - **`@Temporal(TemporalType.TIMESTAMP)`**: se utiliza cuando se quiere **mapear tanto la fecha como la hora**, incluyendo la información de la hora, minutos, segundos y fracciones de segundo.

Utilizar `LocalDate` y `LocalDateTime` con *Java 8+* y frameworks modernos. Recurrir a **`@Temporal` solo en entornos antiguos** (*Java 7* o anteriores).

```
@Entity
public class Evento {
    @Id
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date fechaHora; // Se almacenará la fecha y hora exacta
}
```

- **`@Temporal(TemporalType.DATE)`**: se utiliza **cuando solo se desea almacenar la fecha** (sin información de la hora).

```
@Entity
public class Cumpleanos {
    @Id
    private Long id;
    @Temporal(TemporalType.DATE)
    private Date fecha;
}
```

✓ **`@Enumerated`**: se utiliza para **mapear una enumeración** (enum) **a una columna**. Esta anotación permite especificar cómo deben ser almacenados los valores de la enumeración en la tabla.

- **`EnumType.ORDINAL`**: mapea los **valores de la enumeración como números ordinales** (índices de los valores definidos en la enumeración: 0, 1, 2, etc.).
- **`EnumType.STRING`**: mapea los **valores de la enumeración como cadenas de texto**, utilizando el nombre de cada constante.

```
public enum EstadoPedido {
    PENDIENTE,
    EN_PROCESO,
    TERMINADO
}
@Entity
public class Pedido {
    @Id
    private Long id;
    @Enumerated(EnumType.STRING)
    private EstadoPedido estado;
    // Otros atributos y métodos
}
```

En este ejemplo, se almacenarán los valores como cadenas de texto: `PENDIENTE` se almacena como "PENDIENTE", `EN_PROCESO` se almacena como "EN_PROCESO" y `TERMINADO` se almacena como "TERMINADO".

Una **subclase de entidad hereda todos los atributos persistentes de la entidad de la que extiende**.

Ejemplo: mapeo de una tabla pokemon.

```
CREATE TABLE pokemon (
  id_pokemon INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(15) NOT NULL,
  peso DECIMAL(6, 2) NOT NULL,
  altura DECIMAL(6, 2) NOT NULL,
  ps TINYINT UNSIGNED NOT NULL,
  ataque TINYINT UNSIGNED NOT NULL,
  defensa TINYINT UNSIGNED NOT NULL,
  especial TINYINT UNSIGNED NOT NULL,
  velocidad TINYINT UNSIGNED NOT NULL,
```

Empleado.java

```
import jakarta.persistence.*;

no usages
@Entity
@Table(name = "empleados")
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_empleado")
    private Long id;

    no usages
    @Column(name = "nombre", nullable = false, length = 50)
    private String nombre;

    no usages
    @Column(name = "apellido", nullable = false, length = 50)
    private String apellido;

    no usages
    @Column(name = "salario", precision = 10, scale = 2)
    private double salario;

    no usages
    @Temporal(TemporalType.DATE)
    @Column(name = "fecha_contratacion")
    private java.util.Date fechaContratacion;

    // Otros atributos, constructores, getters y setters
}
```

Empleado.java

```
import jakarta.persistence.*;

1 usage
enum TipoEmpleado {
    no usages
    TIEMPO_COMPLETO,
    no usages
    MEDIO_TIEMPO,
    no usages
    CONTRATISTA
}

no usages
@Entity
@Table(name = "empleados")
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    no usages
    @Column(name = "nombre")
    private String nombre;

    no usages
    @Enumerated(EnumType.STRING)
    @Column(name = "tipo_empleado")
    private TipoEmpleado tipoEmpleado;

    // Otros atributos, constructores, getters y setters
}
```

```
    sexo ENUM('M', 'H', 'MH') NOT NULL,
    descripcion VARCHAR(300)
);
```

Clase *Java* que mapea la tabla pokemon:

```
import jakarta.persistence.*;
import java.math.BigDecimal;

@Entity
@Table(name = "pokemon")
public class Pokemon {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_pokemon")
    private int idPokemon;

    @Column(length = 15, nullable = false)
    private String nombre;

    @Column(nullable = false, precision = 6, scale = 2)
    private BigDecimal peso;

    @Column(nullable = false, precision = 6, scale = 2)
    private BigDecimal altura;

    @Column(nullable = false)
    private short ps;

    @Column(nullable = false)
    private short ataque;

    @Column(nullable = false)
    private short defensa;

    @Column(nullable = false)
    private short especial;

    @Column(nullable = false)
    private short velocidad;

    @Enumerated(EnumType.STRING)
    @Column(name = "sexo", nullable = false)
    private Sexo sexo;

    @Column(length = 300)
    private String descripcion;

    public enum Sexo {M, H, MH}

    // Constructores, getters y setters
    public Pokemon(int idPokemon, String nombre, BigDecimal peso, BigDecimal altura, short ps,
      short ataque, short defensa, short especial, short velocidad, Sexo sexo, String descripcion) {
        this.idPokemon = idPokemon;
        this.nombre = nombre;
        this.peso = peso;
        this.altura = altura;
        this.ps = ps;
        this.ataque = ataque;
        this.defensa = defensa;
        this.especial = especial;
        this.velocidad = velocidad;
        this.sexo = sexo;
        this.descripcion = descripcion;
    }

    public Pokemon() { }

    @Override
    public String toString() {
        return nombre;
    }
}
```

En *JPA*, dependiendo del contexto y los requisitos de la aplicación se pueden usar:

- **Tipos primitivos** si los **valores no pueden ser `null` y el rendimiento es una prioridad**.
- **Wrappers si se necesita manejar valores `null`** o representar de manera explícita la ausencia de valor, especialmente cuando los datos pueden ser opcionales o faltantes.

Tipo de dato *SQL*	Tipo de dato *Java*
TINYINT	byte / Byte
SMALLINT	short / Short
MEDIUMINT	int / Integer
INT o INTEGER	int / Integer
BIGINT	long / Long
DECIMAL(p,s) o NUMERIC	BigDecimal
FLOAT	float / Float
DOUBLE o REAL	double / Double
CHAR(n)	String
VARCHAR(n)	String
TEXT	String
ENUM	enum
SET	String / Set<String>
DATE	java.time.LocalDate
TIME	java.time.LocalTime
DATETIME o TIMESTAMP	java.time.LocalDateTime
YEAR	short / Short
BLOB	byte[] / Byte[]
BOOLEAN o BIT(1)	boolean / Boolean
JSON	String / JsonNode

Para los tipos de datos numéricos UNSIGNED en *SQL*, se debe utilizar en *Java* el tipo de datos superior que garantice cubrir el rango completo de valores cuando sea necesario:

- `TINYINT UNSIGNED` → `short`
- `SMALLINT UNSIGNED` → `int`
- `INT UNSIGNED` → `long`

En *JPA*, **si la precisión no es crítica, se puede usar `double` o `float` en lugar de `BigDecimal`**. Estos tipos son más eficientes en términos de rendimiento y almacenamiento, aunque pueden introducir pequeñas imprecisiones debido a su representación.

El tipo de datos de números de punto flotante más grande en *Java* es `double`. Sin embargo, ¿qué sucede si el número que se necesita es tan grande que ni siquiera cabe en un `double`? Para estos casos, *Java* ofrece la clase especial **`BigDecimal`**, que teóricamente no tienen un tamaño máximo, aunque en la práctica está limitada por la memoria disponible.

A diferencia de los tipos numéricos primitivos, `BigDecimal` no utiliza los operadores matemáticos estándar (+, -, *, /). En su lugar, proporciona un conjunto de métodos específicos para realizar operaciones matemáticas:

```
BigDecimal bd1 = new BigDecimal("4.0");
BigDecimal bd2 = new BigDecimal("2.0");

BigDecimal sum = bd1.add(bd2);
BigDecimal difference = bd1.subtract(bd2);
BigDecimal quotient = bd1.divide(bd2);
BigDecimal product = bd1.multiply(bd2);
```

Se puede usar el método `setScale(int, RoundingMode)` para establecer la cantidad de decimales deseada en un `BigDecimal`. Más información en:
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/math/BigDecimal.html

Ejemplo: uso de `@Embeddable` y `@Embedded`.

```
import jakarta.persistence.Embeddable;
@Embeddable                 // No crea tabla, se usa para marcar la clase como un objeto embebible
public class Direccion {
    private String calle;
    private String ciudad;
    private String codigoPostal;
    // Constructores, getters y setters
}
```

```
import jakarta.persistence.*;
@Entity
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nombre;   // El campo en la tabla tendrá el mismo nombre
    @Embedded                // Indica la clase marcada como @Embeddable que será parte de la tabla
    private Direccion direccion;
    public Persona() {}      // Constructor sin argumentos (necesario para JPA)
    public Persona(String nombre, Direccion direccion) {
        this.nombre = nombre;
        this.direccion = direccion;
    }
    // Getters y setters
}
```

Ejemplo: suponer que se desea modelar entidades de animales en una *BD*. Se podría crear una clase base `Animal` con los atributos comunes, y luego generar clases específicas para cada tipo de animal utilizando `@MappedSuperclass`.

```
import jakarta.persistence.*;
@MappedSuperclass // @MappedSuperclass indica que la clase no se mapeará directamente a una tabla
public class Animal {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(name = "nombre")
    private String nombre;
    // Getters, setters y otros atributos y métodos comunes a todos los animales
}
```

```
import jakarta.persistence.*;
@Entity
@Table(name = "perros")      // Solo es necesario si cambia el nombre de la tabla respecto al modelo
public class Perro extends Animal {
    @Column(name = "raza")
    private String raza;
    // Getters, setters y otros atributos y métodos específicos de los perros
}
```

En este ejemplo, la clase `Perro` hereda los atributos de la clase `Animal` y se mapea a una tabla llamada `perros`. El atributo `nombre` de la clase `Animal` se mapea a la columna `nombre` en la tabla `perros`, mientras que el atributo `raza` se mapea a la columna `raza`.

3.3 Relaciones (asociaciones entre entidades).

En una ***BD* relacional, las relaciones entre registros se implementan mediante claves foráneas**. Este mecanismo asegura que las relaciones sean **bidireccionales**: a partir del valor de una clave foránea, es posible acceder al registro con la clave principal a la que hace referencia, y viceversa.

En la *POO*, sin embargo, **las asociaciones funcionan de manera diferente**. Estas **se implementan mediante referencias entre objetos**. Si un objeto tiene una referencia a otro, puede acceder a este último. Sin embargo, el acceso inverso no está garantizado, a menos que el segundo objeto también contenga una referencia explícita al primero.

Para mapear las relaciones de una *BD*, es necesario crear los atributos correspondientes en las clases que

representen la relación. Estos atributos se mapearán dependiendo de si la relación es unidireccional o bidireccional. Aunque en ambos lados se debe indicar el tipo de relación, solo uno contendrá la información de mapeo. Los **tipos de relaciones más comunes** son:

- ✓ **`@OneToOne`**: representa una **relación 1:1**.
- ✓ **`@ManyToOne`**: indica una **relación N:1**. En este caso, el **atributo corresponde al lado 1**.
- ✓ **`@OneToMany`**: define una **relación 1:N**. En este caso, el **atributo corresponde al lado N**.
- ✓ **`@ManyToMany`**: representa una **relación N:M**. En este tipo de relación, **se especifica una tabla intermedia que mantiene las referencias entre ambas tablas, junto con los campos que actúan como claves foráneas**.

Para establecer una **relación bidireccional**, es **necesario especificar el tipo de relación en ambos lados**. En **uno de los lados**, se utiliza la propiedad **`mappedBy`** **para indicar el atributo de la otra clase que contiene toda la información sobre el mapeo**. Por ejemplo: `@OneToOne(mappedBy = "empleado")`

A continuación, se describen algunas **propiedades comunes de las anotaciones** utilizadas en mapeos de relaciones:

- ✓ **`mappedBy`**: especifica el **atributo en la entidad relacionada que representa la relación**. Se utiliza para **establecer relaciones bidireccionales**.
- ✓ **`cascade`**: define **cómo las operaciones de cambio de estado en la entidad principal afectan a las entidades relacionadas**. Las opciones más comunes son:
  - **`CascadeType.PERSIST`**: al persistir la entidad principal, también se persisten las entidades relacionadas.
  - **`CascadeType.MERGE`**: al fusionar la entidad principal, también se fusionan las entidades relacionadas.
  - **`CascadeType.REMOVE`**: al eliminar la entidad principal, también se eliminan las entidades relacionadas.
  - **`CascadeType.REFRESH`**: al refrescar la entidad principal, también se refrescan las entidades relacionadas.
  - **`CascadeType.DETACH`**: al desvincular la entidad principal, también desvincula las entidades relacionadas.
  - **`CascadeType.ALL`**: aplica todas las operaciones de cambio de estado a las entidades relacionadas.
- ✓ **`optional`**: indica **si la asociación es opcional**. Si se establece en `false`, la entidad relacionada debe existir y no puede ser nula.
- ✓ **`orphanRemoval`**: especifica **si las entidades relacionadas deben eliminarse autmáticamente** (huérfanas) cuando se eliminan de la colección en la entidad principal. Por defecto, está desactivado (`false`). Al establecer `orphanRemoval=true`, se eliminan los objetos dependientes que no tienen una referencia válida al objeto propietario.
- ✓ **`fetch`**: determina **cómo se cargarán los datos de la entidad relacionada**.
  - **`LAZY`** (carga perezosa): los **datos se cargan de manera diferida, solo cuando son necesarios**. Comportamiento **predeterminado para las relaciones** de tipo **`@OneToMany`** y **`@ManyToMany`**.
  - **`EAGER`** (carga ansiosa): los **datos se cargan de forma inmediata junto con la entidad principal**. Comportamiento **predeterminado para las relaciones** de tipo **`@ManyToOne`** y **`@OneToOne`**.

Las **estrategias de fetching en *JPA* son fundamentales para optimizar el rendimiento** de la aplicación y controlar cuándo y cómo se cargan los datos de las relaciones entre entidades.

Lo normal es usar `FetchType.LAZY` por defecto y solo usar `FetchType.EAGER` para relaciones críticas que siempre se necesiten.

Para **personalizar los detalles de las asociaciones entre entidades**, se pueden utilizar las siguientes anotaciones:

- ✓ **`@JoinColumn`**: esta anotación se emplea para **especificar la columna de la clave foránea** que conecta dos entidades en una relación. **Propiedades** principales:
  - **`name`**: define el **nombre de la columna en la tabla que se utiliza para la relación**.
  - **`referencedColumnName`**: **especifica el nombre de la columna en la tabla a la que se hace referencia**. Si no se indica, se asume que se refiere a la **clave primaria de la tabla referenciada**. Ej.: `@JoinColumn(name = "nombre_columna", referencedColumnName = "id_referenciado")`
  - **`nullable`**: **indica si la columna puede contener valores nulos**.
  - **`unique`**: **define si los valores de la columna deben ser únicos**.
- ✓ **`@JoinTable`**: se utiliza **para mapear relaciones muchos a muchos**, donde se necesita una tabla intermedia que almacene la relación entre las entidades. Se coloca en la entidad propietaria (la que tiene la colección de la otra entidad). **Propiedades** principales:
  - **`name`**: especifica el **nombre de la tabla de unión**.
  - **`joinColumns`**: define las **columnas de la tabla actual utilizadas para la unión**.
  - **`inverseJoinColumns`**: define las **columnas de la otra tabla que participa en la unión**.
  - **`uniqueConstraints`**: permite establecer **restricciones únicas en las columnas de la tabla de unión**.
- ✓ **`@PrimaryKeyJoinColumn`**: esta anotación es útil **en relaciones uno a uno cuando se desea que las entidades**

compartan la misma clave primaria. Su **uso es común en esquemas de herencia**, donde las **subclases comparten la clave primaria de la superclase**.

Las anotaciones `@OneToMany`, `@ManyToOne`, `@OneToOne` *y* `@ManyToMany` *en JPA son anotaciones de relación que establecen la naturaleza y el tipo de relación entre entidades. Por otro lado,* `@JoinColumn` *y* `@JoinTable` *son anotaciones que se utilizan para personalizar los detalles de cómo se realiza la asociación en la BD.*

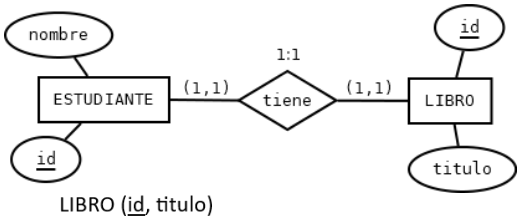
3.3.1 Uno a uno (1:1).

Una asociación uno a uno es **aquella en la que cada objeto de una entidad se asocia con un único objeto de la otra entidad**. Pueden ser **unidireccionales o bidireccionales**. **Se implementan mediante la anotación con `@OneToOne`**.

- **Uno a uno unidireccional.**

En este caso, **una entidad tiene una referencia a la otra, pero no al revés**.

Ejemplo: suponer dos entidades `ESTUDIANTE` y `LIBRO`. Se quiere establecer una asociación donde cada estudiante tenga un libro asignado, pero los libros no necesitan conocer al estudiante. `ESTUDIANTE` es el dueño de la relación.



LIBRO (id, titulo)

ESTUDIANTE (id, nombre, id_libro [fk: LIBRO])

```java
import jakarta.persistence.*;
@Entity
@Table(name = "estudiante")
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(nullable = false, length = 50)
    private String nombre;
    @OneToOne(cascade = CascadeType.ALL, optional = false)
    @JoinColumn(name = "id_libro")     // Especifica la clave foránea (no necesario). Si no
    private Libro libro;               // se pone JPA generaría una por defecto (libro_id)
    public Estudiante() {} // Constructor sin argumentos (requerido por JPA)
    public Estudiante(String nombre, Libro libro) { this.nombre = nombre; this.libro = libro; }
    // Getters y setters
}
```

`CascadeType.ALL` especifica que todas las **operaciones de persistencia realizadas en una entidad se propaguen automáticamente a la entidad relacionada**. Lo habitual es poner `CascadeType.ALL` solo en un lado (el lado propietario de la relación) en las asociaciones bidireccionales para no caer en la cascada recursiva.

```java
import jakarta.persistence.*;
@Entity
@Table(name = "libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(nullable = false, length = 100)
    private String titulo;
    public Libro() {} // Constructor sin argumentos (requerido por JPA)
    public Libro(String titulo) { this.titulo = titulo; }
    // Getters y setters
}
```

En este ejemplo:

- ✓ `@OneToOne` en el atributo `libro` de la clase `Estudiante` establece la relación uno a uno.
- ✓ `cascade = CascadeType.ALL` asegura que las operaciones (guardar, actualizar, eliminar, etc.) realizadas en `Estudiante` se reflejen automáticamente en `Libro`.
- ✓ `@JoinColumn(name = "id_libro")` indica que la columna `id_libro` en la tabla `ESTUDIANTE` actúa como clave foránea.

- **Uno a uno bidireccional.**

En este caso, **ambas entidades tienen referencias entre sí**.

Ejemplo: relación donde `Estudiante` (dueño de la relación) tiene un `Libro`, y el `Libro` también conoce al `Estudiante`.

```java
import jakarta.persistence.*;
```

En las **relaciones unidireccionales** solo se necesita **definir la relación en una entidad**. La referencia existe únicamente en una dirección.

En las **relaciones bidireccionales** ambas entidades conocen la relación, lo que permite navegar en ambas direcciones. Se necesita **definir la relación en ambas entidades y usar la propiedad `@mappedBy` en una de ellas**.

`@mappedBy` asegura que no se cree una columna adicional de clave foránea.


```java
@Entity
@Table(name = "estudiante")
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String nombre;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "id_libro")
    private Libro libro;

    public Estudiante() { }

    public Estudiante(String nombre, Libro libro) {
        this.nombre = nombre;
        this.libro = libro;
        libro.setEstudiante(this); // Establece la relación en ambas direcciones
    }

    // Getters y setters
}
```

Si **no se incluye `@JoinColumn` *JPA* asigna un nombre automáticamente a la clave foránea** que está formado por el nombre de la tabla a la que hace referencia, más _ e `id`. Ejemplo: `libro_id`, `curso_id`, etc.

En *JPA*, el **dueño de la relación es la entidad que gestiona la clave foránea** en la *BD*. Esto **se indica mediante la anotación `@JoinColumn`**, que se usa en el lado propietario de la relación para especificar la columna de clave foránea.

Por otro lado, el **atributo `mappedBy` se utiliza en el lado no propietario de la relación** para señalar que la relación está definida en el otro lado. Esto significa que el lado con `mappedBy` no contiene la clave foránea, sino que refleja la relación gestionada por el otro lado.

```java
import jakarta.persistence.*;
@Entity
@Table(name = "libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String titulo;

    @OneToOne(mappedBy = "libro", fetch = FetchType.LAZY, optional = false)
    private Estudiante estudiante;

    public Libro() { }

    public Libro(String titulo) { this.titulo = titulo; }

    // Getters y setters
    public void setEstudiante(Estudiante estudiante) { this.estudiante = estudiante; }

    // ...
}
```

En este ejemplo:

- ✓ En la clase `Estudiante`, se usa `@OneToOne @JoinColumn(name="id_libro")` para especificar que `id_libro` en la tabla `ESTUDIANTE` es la clave foránea.
- ✓ En la clase `Libro`, se utiliza `@OneToOne(mappedBy="libro")` para indicar que la relación está definida en el atributo `libro` de la clase `Estudiante`.
- ✓ En el constructor de `Estudiante`, se asegura de establecer la relación en ambas direcciones con `libro.setEstudiante(this)`.

➢ **Compartir el mismo `ID` por las dos entidades en una relación 1:1 unidireccional/bidireccional.**

- Clase `Libro` igual.
- Clase `Estudiante`:

```java
import jakarta.persistence.*;
@Entity
@Table(name = "estudiante")
public class Estudiante {
    @Id
    // @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 50)
    private String nombre;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "id_libro")
    @MapsId
    private Libro libro;

    public Estudiante(String nombre, Libro libro) { this.nombre = nombre; this.libro = libro; }
    public Estudiante() {}
    // Getters y setters
}
```

La **anotación `@MapsId`** es especialmente útil en relaciones uno a uno (`@OneToOne`) donde **ambas entidades comparten la misma clave primaria**.

La relación entre las entidades se realiza mediante una columna que actúa como clave primaria y clave foránea al mismo tiempo.

La anotación `@MapsId` siempre se coloca en el lado propietario de la relación en *JPA*.

➢ **Comparación entre asociación unidireccional y bidireccional:**

Aspecto	Unidireccional	Bidireccional
Dirección de acceso	De una entidad hacia otra.	Acceso desde ambas entidades.
Anotación mappedBy	No se utiliza.	Requerida para vincular la relación.
Mantenimiento de datos	Menos complejo.	Puede ser más complejo de mantener.
Ejemplo de uso	Relaciones simples y limitadas.	Relaciones donde ambos lados interactúan.

3.3.2 Uno a muchos (1:N) y muchos a uno (N:1).

Para modelar una asociación uno a muchos (1:N), se requiere que el objeto principal incluya una colección para almacenar las referencias a los objetos relacionados. Estas **colecciones deben ser de tipo `List`, `Map` o `Set`**, ya que son las únicas permitidas en este contexto.

Las **anotaciones** principales utilizadas son:

- ✓ **`@OneToMany`**: se **aplica al atributo que contiene la colección y admite el parámetro `cascade`**.
- ✓ **`@ManyToOne`**: se utiliza **en el lado "muchos" de la relación para definir la asociación con la entidad principal**.

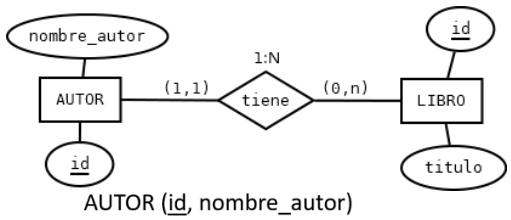
Es importante destacar que una asociación `@OneToMany` es equivalente a una asociación `@ManyToOne` en sentido inverso. En relaciones bidireccionales, las **dos asociaciones unidireccionales deben vincularse utilizando el parámetro `mappedBy`**. Sin embargo, la **anotación `@ManyToOne` no admite el uso de `mappedBy`, por lo que esta configuración debe especificarse siempre en el lado `@OneToMany`**.

Ejemplo: suponer que un autor puede tener varios libros, pero cada libro está asociado a un único autor.

➢ **Unidireccional**: la entidad `Autor` mantendrá una lista de libros, pero la entidad `Libro` no tendrá referencia al autor (uno a muchos unidireccional).

Uno a muchos unidireccional

```java
import jakarta.persistence.*;
import java.util.List;
import java.util.ArrayList;
@Entity
@Table(name = "autor")
public class Autor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(name = "nombre_autor", nullable = false, length = 50)
    private String nombre;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_autor") // *** FK tabla Libro ***
    private List<Libro> libros = new ArrayList<>();
    public Autor() {}
    public Autor(String nombre) { this.nombre = nombre; }
    // Getters y setters
    public void agregarLibro(Libro libro) {libros.add(libro);}
}
```



AUTOR (id, nombre_autor)

LIBRO (id, titulo, id_autor [fk: AUTOR])

Situación poco intuitiva, pero si no se incluye genera una tabla intermedia y hay que realizar más operaciones con lo que se degrada el rendimiento.

```java
import jakarta.persistence.*;
@Entity
@Table(name = "libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(nullable = false, length = 50)
    private String titulo;
    public Libro() {}
    public Libro(String titulo) { this.titulo = titulo; }
    // Getters y setters
}
```

Muchos a uno unidireccional

```java
import jakarta.persistence.*;
@Entity
@Table(name = "autor")
public class Autor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(name = "nombre_autor", nullable = false, length = 50)
    private String nombre;
    public Autor() {}
    public Autor(String nombre) { this.nombre = nombre; }
    // Getters y setters
}
```

```java
import jakarta.persistence.*;
@Entity
@Table(name = "libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String titulo;
    @ManyToOne
    @JoinColumn(name = "id_autor")
    private Autor autor;
    public Libro() {}
    public Libro(String titulo, Autor autor) {
        this.titulo = titulo;
        this.autor = autor;
    }
    // Getters y setters
}
```

➢ **Bidireccional**: la clase `Autor` contiene una lista de libros, mientras que cada `Libro` tiene una referencia a su autor.

```java
import jakarta.persistence.*;
import java.util.List;
```


```java
import java.util.ArrayList;
@Entity
@Table(name = "autor")
public class Autor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(name = "nombre_autor", nullable = false, length = 50)
    private String nombre;
    @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL)
    private List<Libro> libros = new ArrayList<>();
    public Autor() {}        // Constructor sin argumentos (requerido por JPA)
    public Autor(String nombre) { this.nombre = nombre; }
    public void agregarLibro(Libro libro) {
        libros.add(libro);
        libro.setAutor(this);       // Establecer la relación bidireccional
    }
    // Getters y setters
}

import jakarta.persistence.*;
@Entity
@Table(name = "libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String titulo;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_autor")
    private Autor autor;
    public Libro() {}        // Constructor sin argumentos (necesario para JPA)
    public Libro(String titulo, Autor autor) {
        this.titulo = titulo;
        this.autor = autor;
    }
    public void setAutor(Autor autor) {
        this.autor = autor;
    }
    // Getters y setters
}
```

Para **evitar problemas en relaciones bidireccionales, utilizar cascadas solo en el lado necesario** para evitar operaciones descontroladas. Además, al mostrar las entidades, **controlar manualmente la impresión**: evitar incluir relaciones bidireccionales en el método `toString()` para prevenir recursividad infinita y errores como `StackOverflowError`.

En la clase `Autor`, el atributo `libros` es una colección (`List`) que define la asociación 1:N entre los autores y sus libros. Esta relación se especifica como bidireccional mediante `mappedBy`, que indica que el mapeo se encuentra en el atributo `autor` de la entidad `Libro`. Además, se utiliza `CascadeType.ALL` para que todas las operaciones en un autor (persistir, eliminar y actualizar) se propaguen automáticamente a los libros asociados.

En la clase `Libro`, se utiliza la anotación `@ManyToOne` para indicar la relación con el autor. La anotación `@JoinColumn` personaliza el nombre de la columna que actúa como clave foránea (`id_autor`) en la tabla `LIBRO`. Si esta anotación se elimina, la relación sería unidireccional, y habría que eliminar el `mappedBy` en la anotación `@OneToMany` de la clase `Autor`.

3.3.3 Muchos a muchos (N:M).

En una asociación muchos a muchos, **muchas instancias de una entidad están asociadas con muchas instancias de otra entidad**. Para mapear este tipo de asociación en *JPA*, se utiliza la **anotación `@ManyToMany`**.

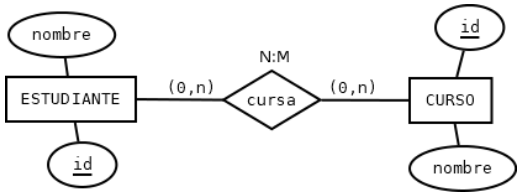
Ejemplo: suponer que se tienen dos tablas `ESTUDIANTE` y `CURSO`. Cada estudiante puede estar inscrito en varios cursos, y cada curso puede tener varios estudiantes inscritos.

```java
import jakarta.persistence.*;
import java.util.List;
import java.util.ArrayList;
@Entity
@Table(name = "estudiante")
```

En las *BD*, las **relaciones N:M se modelan mediante una tabla intermedia** que contiene las claves primarias de ambas entidades.

```java
public class Estudiante {    // Dueño de la relación
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 50)
    private String nombre;

    @ManyToMany
    @JoinTable(
        name = "cursa",      // Tabla intermedia
        joinColumns = @JoinColumn(name = "id_estudiante"),      // Clave foránea de ESTUDIANTE
        inverseJoinColumns = @JoinColumn(name = "id_curso")     // Clave foránea de CURSO
    )
    private Set<Curso> cursos = new HashSet<>();

    public Estudiante() {} // Constructor sin argumentos (necesario para JPA)
    public Estudiante(String nombre) { this.nombre = nombre; }

    // Getters y setters
}
```

En **relaciones N:M**, generalmente se recomienda **usar Set porque asegura unicidad (sin duplicados), y es más eficiente en búsquedas y operaciones** (agrega clave primaria a la tabla que no permite valores duplicados). Sin embargo, si se necesita mantener un orden específico o acceder a elementos por índice, se puede optar por `List`, aunque esto permitiría duplicados.



ESTUDIANTE (id, nombre)

CURSO (id, nombre)

CURSA (id_estudiante [fk: ESTUDIANTE], id_curso [fk: CURSO])

```java
import jakarta.persistence.*;
import java.util.List;
import java.util.ArrayList;

@Entity
@Table(name = "curso")
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 50)
    private String nombre;

    @ManyToMany(mappedBy = "cursos")
    private Set<Estudiante> estudiantes = new HashSet<>();

    // Constructor sin argumentos (necesario para JPA)
    public Curso() {}
    public Curso(String nombre) { this.nombre = nombre; }

    // Getters y setters
}
```

Si no se agrega esta parte la asociación sería unidireccional, si se agrega pasa a ser bidireccional.

En este ejemplo:

- ✓ En la entidad `Estudiante`, se usa `@ManyToMany` para indicar una relación muchos a muchos con la entidad `Curso`.
- ✓ En la entidad `Curso`, se usa `@ManyToMany(mappedBy = "cursos")` para indicar que la relación está mapeada por el atributo `cursos` en la entidad `Estudiante`.
- ✓ `@JoinTable` se utiliza para especificar la tabla de unión que almacenará la relación. Se definen las columnas de unión usando `joinColumns` e `inverseJoinColumns`.

Ejemplo: **relación N:M con atributos**. Suponer que se quiere almacenar la fecha de inscripción de cada estudiante en un curso.

En algunos casos, la tabla intermedia tiene atributos adicionales (por ejemplo, la fecha de inscripción). Para modelar esto, **se debe crear una clase específica para representar la tabla intermedia**.

```java
import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "cursa")
public class Cursa {
    @Id
    @ManyToOne
    @JoinColumn(name = "id_estudiante", nullable = false)
    private Estudiante estudiante;

    @Id
    @ManyToOne
    @JoinColumn(name = "id_curso", nullable = false)
    private Curso curso;

    private LocalDate fechaInscripcion;

    // Constructores, getters y setters
}
```

```java
import jakarta.persistence.*;
import java.io.Serializable;

@Embeddable
public class CursaId implements Serializable {
    private Integer estudianteId;
    private Integer cursoId;

    // Constructor, getters y setters
}
```

```java
import jakarta.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "cursa")
public class Cursa {
    @EmbeddedId
    private CursaId id;

    @ManyToOne
    @MapsId("estudianteId")
    @JoinColumn(name = "id_estudiante", nullable = false)
    private Estudiante estudiante;

    @ManyToOne
    @MapsId("cursoId")
    @JoinColumn(name = "id_curso", nullable = false)
    private Curso curso;

    private LocalDate fechaInscripcion;

    // Constructor, getters y setters
}
```

Forma más correcta para Cursa.

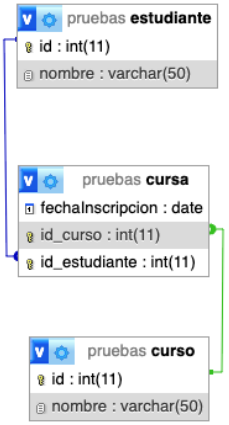
```java
import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "curso")
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 50)
    private String nombre;

    @OneToMany(mappedBy = "curso", cascade = CascadeType.ALL)
    private List<Cursa> inscripciones = new ArrayList<>();

    // Constructores, getters y setters
}
```

```java
import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "estudiante")
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 50)
    private String nombre;

    @OneToMany(mappedBy = "estudiante", cascade = CascadeType.ALL)
    private List<Cursa> inscripciones = new ArrayList<>();

    // Constructores, getters y setters
}
```

```java
import java.io.Serializable;
public class CursaId implements Serializable {
  private Integer estudiante;
  private Integer curso;

  // Constructor, getters, setters, equals y hashCode
}
```

```java
import jakarta.persistence.*;
import java.time.LocalDate;
@Entity
@Table(name = "cursa")
@IdClass(CursaId.class)
public class Cursa {
  @Id
  @ManyToOne
  @JoinColumn(name = "id_estudiante", nullable = false)
  private Estudiante estudiante;

  @Id
  @ManyToOne
  @JoinColumn(name = "id_curso", nullable = false)
  private Curso curso;

  private LocalDate fechaInscripcion;

  // Constructores, getters y setters
}
```

Otra forma más para Cursa.



3.4 Herencia.

En *JPA*, se puede utilizar la herencia para **modelar relaciones del tipo "es un/a" o "es un tipo de"**. La anotación **`@Inheritance` configura cómo se mapeará la jerarquía de herencia en la *BD* mediante el parámetro `strategy`**. Si no se especifica, se usará la estrategia **`SINGLE_TABLE` por defecto**.

`@DiscriminatorColumn` configura la **columna discriminadora que distingue las subclases** y **`@DiscriminatorValue`** asigna un **valor único para identificar cada subclase en la columna discriminadora**.

Estrategias de herencia disponibles:

- **`SINGLE_TABLE`**: en esta estrategia, todas las **clases de la jerarquía comparten una única tabla** en la *BD*. Una **columna adicional permite distinguir entre los diferentes tipos de entidades**.

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo_entidad", discriminatorType = DiscriminatorType.STRING)
public class EntidadBase {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;
    protected String atributoComun;
}

@Entity
@DiscriminatorValue("TIPO_A")
public class TipoA extends EntidadBase {
    protected String atributoTipoA;
}

@Entity
@DiscriminatorValue("TIPO_B")
public class TipoB extends EntidadBase {
    protected String atributoTipoB;
}
```

```sql
CREATE TABLE entidad_base (
    id INT PRIMARY KEY,
    tipo_entidad VARCHAR(255),
    atributoComun VARCHAR(255),
    atributoTipoA VARCHAR(255),
    atributoTipoB VARCHAR(255)
);
```

Recordar que la anotación **`@MappedSuperclass` se utiliza para definir una clase base cuyos atributos se heredan en las entidades que la extienden**. Sin embargo, una **clase anotada con `@MappedSuperclass` no se mapea directamente a una tabla** en la *BD*. Es útil para factorizar propiedades comunes en múltiples entidades.

En este caso, `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)` indica el uso de la estrategia de herencia de tabla única. La columna `tipo_entidad` distingue entre `TipoA` y `TipoB`.

- **TABLE_PER_CLASS**: con esta estrategia, cada **clase hija** (subclase) **en la jerarquía tiene su propia tabla**, **que incluye todos los atributos** (propios y heredados). No hay una tabla común para toda la jerarquía.

```java
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class EntidadBase {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;

    protected String atributoComun;
}

@Entity
public class TipoA extends EntidadBase {
    protected String atributoTipoA;
}

@Entity
public class TipoB extends EntidadBase {
    protected String atributoTipoB;
}
```

```sql
CREATE TABLE tipo_a (
    id INT PRIMARY KEY,
    atributoComun VARCHAR(255),
    atributoTipoA VARCHAR(255)
);

CREATE TABLE tipo_b (
    id INT PRIMARY KEY,
    atributoComun VARCHAR(255),
    atributoTipoB VARCHAR(255)
);
```

En TABLE_PER_CLASS no se requiere una columna discriminatoria.

Aquí, `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)` indica que cada clase hija (`TipoA`, `TipoB`) tendrá su propia tabla con todos los atributos.

- **JOINED**: en esta estrategia, **cada clase de la jerarquía tiene su propia tabla que contiene el identificador y los atributos propios, no los heredados**. Los **atributos comunes se guardan en la tabla base** y las consultas se realizan mediante uniones (`JOIN`).

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class EntidadBase {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;

    protected String atributoComun;
}

@Entity
// @PrimaryKeyJoinColumn(name = "tipo_a_id")
public class TipoA extends EntidadBase {
    protected String atributoTipoA;
}

@Entity
// @PrimaryKeyJoinColumn(name = "tipo_b_id")
public class TipoB extends EntidadBase {
    protected String atributoTipoB;
}
```

```sql
CREATE TABLE entidad_base (
    id INT PRIMARY KEY,
    atributoComun VARCHAR(255)
);

CREATE TABLE tipo_a (
    id INT PRIMARY KEY,
    atributoTipoA VARCHAR(255),
    FOREIGN KEY (id) REFERENCES entidad_base(id)
);

CREATE TABLE tipo_b (
    id INT PRIMARY KEY,
    atributoTipoB VARCHAR(255),
    FOREIGN KEY (id) REFERENCES entidad_base(id)
);
```

Personalizar la clave primaria.

En JOINED las tablas de las subclases tienen una clave foránea que referencia la tabla de la superclase.

En este caso, la información común se almacena en la tabla `EntidadBase` (superclase), mientras que los atributos específicos de `TipoA` y `TipoB` se guardan en sus respectivas tablas.

3.5 Relaciones opcionales.

En *JPA*, las relaciones opcionales **permiten especificar si una entidad relacionada puede ser nula o no**. Esto se controla mediante la **propiedad `optional` en las anotaciones `@ManyToOne`** (para relaciones "muchos a uno") **y `@OneToOne`** (para relaciones "uno a uno"):

- ✓ Relación opcional (**`optional = true`**): permite que el atributo de la relación pueda ser `null`.
- ✓ Relación no opcional (**`optional = false`**): indica que el atributo debe tener siempre un valor no nulo.

Cuando no se especifica la propiedad `optional`, el valor predeterminado es true, lo que significa que la relación es opcional.

- **Relación "muchos a uno" opcional**: suponer que se tiene una entidad `Empleado` con una relación “muchos a uno” con la entidad `Departamento`, y que un empleado puede o no estar asignado a un departamento.

```java
@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```

```java
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
```

```
    private String nombre;
    // Relación "muchos a uno" opcional con Departamento
    @ManyToOne(optional = true)     // Se permite null (opcional)
    @JoinColumn(name = "departamento_id")
    private Departamento departamento;

    // Getters y setters, constructores, etc.
}
```

En este caso, `@ManyToOne(optional = true)` indica que la relación es opcional, permitiendo que el atributo departamento sea nulo.

Nota: la propiedad `optional = true` es el valor por defecto y podría omitirse.

- **Relación "uno a uno" no opcional**: suponer que se tiene una entidad `Estudiante` con una relación "uno a uno" con la entidad `Direccion`, y que cada estudiante debe tener obligatoriamente una dirección asignada.

```
@Entity
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String nombre;

    // Relación "uno a uno" no opcional con Dirección
    @OneToOne(optional = false)     // No se permite null (no opcional)
    @JoinColumn(name = "direccion_id")
    private Direccion direccion;

    // Getters y setters, constructores, etc.
}
```

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
```

En este caso, `@OneToOne(optional=false)` asegura que el atributo `direccion` no puede ser nulo, obligando a que cada estudiante tenga una dirección.

Si se desea que la relación "uno a uno" sea opcional, solo se necesita establecer `optional=true` o simplemente omitir la propiedad, ya que el valor por defecto es `true`.

3.6 Relaciones reflexivas.

En *JPA*, una relación reflexiva **se refiere a una relación en la que una entidad se relaciona consigo misma**. Estas relaciones **pueden ser de diferentes cardinalidades**, como uno a uno (1:1), uno a muchos (1:N) o muchos a muchos (N:M). Por ejemplo, un empleado puede tener un "jefe", que es otro empleado dentro de la misma entidad.

Ejemplo: un empleado tiene un jefe (1:1).

id	nombre	jefe_id
1	Ana	NULL
2	Carlos	1
3	Beatriz	1

```
import jakarta.persistence.*;

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

    private String nombre;

    @OneToOne
    @JoinColumn(name = "jefe_id", nullable = true)       // Columna que almacena el ID del jefe
    private Empleado jefe;          // nullable = true --> Un empleado puede no tener jefe

    // Getters y setters
}
```

Ejemplo: jerarquía de empleados, donde cada empleado puede tener un jefe y múltiples subordinados (1:N).

id	nombre	jefe_id
1	Ana	NULL
2	Carlos	1
3	Beatriz	1
4	David	2

```
import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

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

    private String nombre;

    @ManyToOne
    @JoinColumn(name = "jefe_id", nullable = true)       // FK hacia el jefe
    private Empleado jefe;
```

```
    @OneToMany(mappedBy = "jefe", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Empleado> subordinados = new ArrayList<>();

    // Constructor, getters y setters

    public void agregarSubordinado(Empleado subordinado) {
        subordinados.add(subordinado);
        subordinado.setJefe(this);
    }

    public void eliminarSubordinado(Empleado subordinado) {
        subordinados.remove(subordinado);
        subordinado.setJefe(null);
    }
}
```

> **Por defecto**, las **claves foráneas generadas con `@JoinColumn` son `nullable=true`**, pero ser explícito mejora la claridad del código.

Ejemplo: varios usuarios pueden ser amigos entre sí (N:M).

```
import jakarta.persistence.*;
import java.util.List;

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

    private String nombre;

    @ManyToMany
    @JoinTable(
        name = "amigos",
        joinColumns = @JoinColumn(name = "usuario_id"),
        inverseJoinColumns = @JoinColumn(name = "amigo_id")
    )
    private List<Usuario> amigos;

    // Getters y setters
}
```

Usuario

id	nombre
1	Ana
2	Carlos
3	Beatriz
4	David

amigos

usuario_id	amigo_id
1	2
1	3
2	1
2	4
3	1

3.7 Borrado de objetos huérfanos.

El borrado de objetos huérfanos **se refiere a la eliminación automática de entidades secundarias (relacionadas) cuando** (se declara en los extremos de las relaciones con cardinalidad 1):

- ✓ **La relación entre dos entidades se modifica**.
- ✓ **La entidad principal es eliminada**.

> El **valor por defecto** del atributo **`orphanRemoval` es `false`.**

En *JPA*, **este comportamiento se controla mediante el parámetro `orphanRemoval`**, el cual define si los objetos huérfanos (entidades desconectadas de una relación) deben mantenerse o eliminarse:

- ✓ **`orphanRemoval=false`**: los objetos **huérfanos permanecen en la *BD***.
- ✓ **`orphanRemoval=true`**: los objetos **huérfanos se eliminan automáticamente**.

El uso de **`orphanRemoval` es válido únicamente en relaciones** (no se aplica a los otros tipos de relaciones):

- ✓ Uno a uno (**`@OneToOne`**).
- ✓ Uno a muchos (**`@OneToMany`**).

En ciertas situaciones, el **borrado en cascada (`cascade=CascadeType.REMOVE`) y el borrado de objetos huérfanos (`orphanRemoval=true`) pueden superponerse**, haciendo que el **uso combinado sea redundante**. Por ejemplo, especificar `cascade=CascadeType.REMOVE` ya asegura la eliminación de las entidades relacionadas, incluso sin `orphanRemoval=true`.

Ejemplo:

```
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
```

```
@Entity
public class Padre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @OneToOne(mappedBy = "padre", cascade = CascadeType.ALL, orphanRemoval = true)
    private Hijo hijo;

    // Otros atributos y métodos
}

@Entity
public class Hijo {
    @Id
```

> Al atributo **`cascade`** se le pueden **asignar varios valores** a la vez de la siguiente forma:
>
> **`cascade = {CascadeType.PERSIST, CascadeType.MERGE}`**

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @OneToOne
    private Padre padre;
    // Otros atributos y métodos
}
```

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
```

En este ejemplo, al establecer `orphanRemoval=true` en la anotación `@OneToOne`, se le está diciendo que elimine el objeto `Hijo` si se elimina o se desvincula del objeto `Padre`.

3.8 Eliminaciones en cascada a nivel de *BD*.

En *JPA*, las **anotaciones `@OnDelete` y `@OnUpdate` son proporcionadas por *Hibernate* para gestionar comportamientos específicos en cascada directamente a nivel de la *BD***.

La anotación **`@OnDelete` permite delegar al motor de la *BD* la eliminación de entidades relacionadas mediante el uso de la funcionalidad "`ON DELETE CASCADE`"**. Esto asegura que cuando una entidad principal sea eliminada, las entidades asociadas (hijas o relacionadas) también sean eliminadas automáticamente, sin necesidad de cargarlas en la memoria de la aplicación. Esto **mejora el rendimiento** al reducir la cantidad de operaciones realizadas.

> **Evitar depender de `@OnUpdate`**, ya que las **actualizaciones en cascada son menos comunes y pueden ser complejas**. Mejor manejar este comportamiento explícitamente con lógica en la aplicación o usando `CascadeType.ALL` donde sea necesario.

Ejemplo: modelar y gestionar una relación uno a muchos (`@OneToMany`) entre dos entidades `Padre` e `Hijo`, utilizando la anotación `@OnDelete` para habilitar la eliminación en cascada a nivel de *BD*.

```
import jakarta.persistence.*;
import org.hibernate.annotations.OnDelete;
import org.hibernate.annotations.OnDeleteAction;
import java.util.ArrayList;
import java.util.List;
@Entity
public class Padre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    @OneToMany(mappedBy = "padre", cascade = CascadeType.ALL, orphanRemoval = true)
    @OnDelete(action = OnDeleteAction.CASCADE)
    private List<Hijo> hijos = new ArrayList<>();
    // Constructor, Getters y Setters
    public Padre() {}
    public Padre(String nombre) { this.nombre = nombre; }
    public void agregarHijo(Hijo hijo) { hijos.add(hijo); hijo.setPadre(this); }
}

import jakarta.persistence.*;
@Entity
public class Hijo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    @ManyToOne
    @JoinColumn(name = "padre_id", nullable = false)
    private Padre padre;
    // Constructor, Getters y Setters
    public Hijo() {}
    public Hijo(String nombre) { this.nombre = nombre; }
    public void setPadre(Padre padre) { this.padre = padre; }
}
```

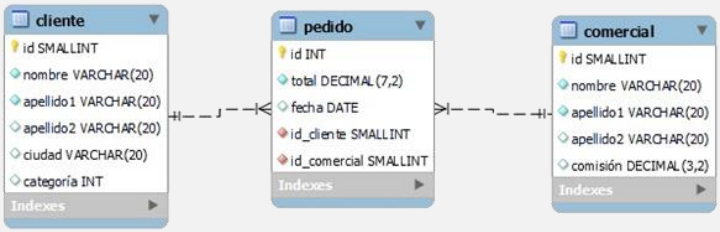
El esquema generado en una *SGBD* como *MariaDB* sería similar al siguiente:

```
CREATE TABLE Padre (
    id BIGINT AUTO_INCREMENT NOT NULL,
    nombre VARCHAR(255),
```

```
    PRIMARY KEY (id)
);
CREATE TABLE Hijo (
    id BIGINT AUTO_INCREMENT NOT NULL,
    nombre VARCHAR(255),
    padre_id BIGINT NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT FK_PADRE_HIJO FOREIGN KEY (padre_id) REFERENCES Padre (id) ON DELETE CASCADE
);
```

TAREA

1. **Crear una clase `Alumno` con los siguientes atributos: `idAlumno` (que identificará cada uno de los alumnos), `nombre`, `direccion` (opcional), `fechaNacimiento` y `notaMedia`. Convertir esta clase en una entidad y hacer que se asigne un `id` distinto de forma automática a cada alumno. Crear la unidad de persistencia (`persistence.xml`) que permita usar una *BD* "alumnos" para almacenar objetos de la clase `Alumno`.**
2. **Realizar el mapeo de las tablas que se muestran en la siguiente imagen. Crear la unidad de persistencia.**



3. **Se desea modelar un sistema para gestionar una biblioteca utilizando *JPA*. En este sistema existan cuatro entidades principales: `Autor`, `Editorial`, `Libro` y `Biblioteca`, con las siguientes relaciones:**
   - **Relación 1:N entre `Autor` y `Libro`: un autor puede escribir varios libros, pero cada libro tiene un único autor.**
   - **Relación 1:1 entre `Libro` y `Editorial`: cada libro pertenece a una única editorial, y una editorial publica un libro por separado (relación simplificada).**
   - **Relación N:M entre `Libro` y `Biblioteca`: un libro puede estar disponible en varias bibliotecas, y cada biblioteca puede tener múltiples libros.**

   Crear las siguientes tablas en una *BD* llamada `gestion_biblioteca`:
   - **`autor`: almacena información sobre los autores.**
   - **`editorial`: almacena información sobre las editoriales.**
   - **`libro`: almacena información sobre los libros.**
   - **`biblioteca`: almacena información sobre las bibliotecas.**
   - **`libro_biblioteca`: tabla intermedia para gestionar la relación muchos a muchos entre libro y biblioteca.**

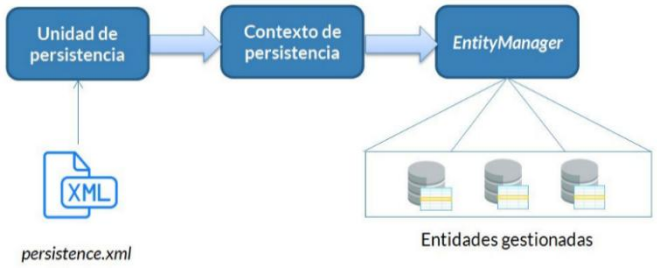
   Implementar las clases de entidad con las siguientes propiedades:
   - **`Autor`:**
     - **`id`: identificador único (clave primaria).**
     - **`nombre`: nombre del autor (máximo 100 caracteres).**
     - **Relación 1:N con `Libro`.**
   - **`Editorial`:**
     - **`id`: identificador único (clave primaria).**
     - **`nombre`: nombre de la editorial (máximo 100 caracteres).**
     - **Relación 1:1 con `Libro`.**
   - **`Libro`:**
     - **`id`: identificador único (clave primaria).**
     - **`titulo`: título del libro (máximo 100 caracteres).**
     - **Relación N:1 con `Autor`.**
     - **Relación 1:1 con `Editorial`.**
     - **Relación N:M con `Biblioteca`.**
   - **`Biblioteca`:**
     - **`id`: identificador único (clave primaria).**
     - **`nombre`: nombre de la biblioteca (máximo 100 caracteres).**
     - **Relación N:M con `Libro`.**

   Implementar los mapeos de relaciones en *JPA*:
   - **Usar `@OneToMany`, `@ManyToOne`, `@OneToOne` y `@ManyToMany` según corresponda.**
   - **Crear la tabla intermedia `libro_biblioteca` para la relación N:M.**

   Proporcionar constructores, getters y setters para cada entidad.

4 Operaciones sobre la *BD*.

4.1 Contexto de persistencia.

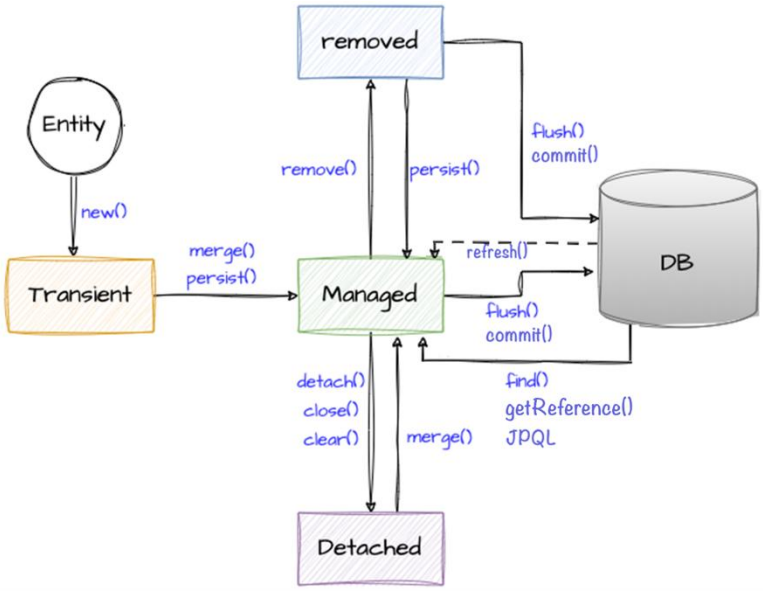


El contexto de persistencia en *JPA* es el **entorno donde se gestionan y mantienen las entidades persistentes de una aplicación**, y **permite realizar operaciones como almacenar, actualizar, eliminar y recuperar datos desde y hacia la *BD***.

El `EntityManager` es el **componente principal de *JPA* encargado de mediar entre el programa y la *BD*, gestionando las operaciones en el contexto de persistencia**.

Una **unidad de persistencia define el conjunto de entidades que pueden ser mapeadas a una *BD*, así como la configuración necesaria para establecer la conexión con ella**.

Las **entidades gestionadas en *JPA* pueden pasar por diferentes estados** durante su ciclo de vida:



- ✓ `transient` (nueva): la **entidad ha sido creada, pero aún no está asociada al contexto de persistencia**. No tiene representación en la *BD*.
- ✓ **managed** (persistente): la **entidad tiene un identificador y está asociada al contexto de persistencia**. Puede estar almacenada en la *BD* o estar pendiente de serlo.
- ✓ **detached** (desconectada): la **entidad tiene un identificador, pero ya no está asociada al contexto de persistencia**, generalmente porque este se ha cerrado.
- ✓ `removed` (eliminada): la **entidad tiene un identificador y está asociada al contexto de persistencia, pero está programada para ser eliminada**.

Para **obtener una instancia del `EntityManager`**, se utiliza la siguiente estructura:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("unidad_persistencia");
EntityManager em = emf.createEntityManager();
```

Al finalizar, es importante **cerrar estas instancias para liberar recursos**:

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

Métodos principales de `EntityManager`:

Método	Descripción
`void persist(Object o)`	Asocia una entidad nueva al contexto de persistencia, almacenándola en la BD.
`void remove(Object o)`	Elimina una entidad tanto del contexto de persistencia como de la BD.
`T find(Class<T> c, Object o)`	Busca una entidad por su clave primaria e inicializa sus datos.
`T getReference(Class<T> c, Object o)`	Obtiene una referencia diferida de una entidad; no accede a la BD hasta que sea necesario.
`Query createQuery(String s)`	Crea una consulta en JPQL.
`TypedQuery<T> createQuery(String s, Class<T> c)`	Crea una consulta tipada en JPQL.
`void refresh(Object o)`	Recarga los datos de una entidad desde la BD.
`T merge(T e)`	Fusiona los cambios de una entidad con el contexto de persistencia, si no existe la crea.
`void flush()`	Sincroniza el contexto de persistencia con la BD.
`void detach(Object o)`	Desvincula una entidad del contexto de persistencia.
`void clear()`	Desvincula todas las entidades gestionadas por el EntityManager.

`boolean contains(Object o)`	Comprueba si una entidad está gestionada por el contexto de persistencia.
`EntityTransaction getTransaction()`	Obtiene la transacción para gestionar las operaciones.
`void close()`	Cierra y libera los recursos asociados al EntityManager.

Consideraciones importantes:

- ✓ La **gestión eficiente del contexto de persistencia es crucial para garantizar la consistencia e integridad de los datos**.
- ✓ **No se permite modificar el identificador de una entidad gestionada**. Si es necesario cambiarlo, se debe eliminar la entidad y crear una nueva con los datos requeridos.

4.2 Manejo de transacciones.

En *JPA*, **cualquier operación que modifique la *BD* debe realizarse dentro de una transacción**. Para ello, **se utiliza el método `getTransaction()` que proporciona acceso a una instancia de `EntityTransaction`**. Esta interfaz representa una transacción y permite realizar operaciones atómicas en la *BD*.

```
EntityTransaction tx = em.getTransaction();
```

Métodos principales de **`EntityTransaction`**:

Método	Descripción
`void begin()`	Inicia una nueva transacción.
`void commit()`	Confirma (realiza un commit) los cambios en la BD.
`void rollback()`	Revierte (hace rollback) los cambios de la transacción.
`boolean isActive()`	Comprueba si la transacción está activa.

Sintaxis:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("unidad_persistencia");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try {
    tx.begin(); // Inicia la transacción
    // Realizar operaciones sobre la BD
    ...
    tx.commit(); // Confirma la transacción
} catch (Exception e) {
    if (tx != null && tx.isActive())
        tx.rollback(); // Revierte la transacción en caso de error
    System.out.println("Error: " + e.getMessage());
} finally {
    if (em != null) em.close();
    if (emf != null) emf.close();
}
```

Para **usar transacciones**, se debe **importar la siguiente clase**:

```
import jakarta.persistence.EntityTransaction;
```

4.3 Persistir.

4.3.1 Objetos.

En una transacción, después de utilizar el método `persist()` para hacer que las entidades sean gestionadas, se puede invocar a `flush()` para sincronizar los cambios pendientes con la *BD* antes de finalizar la transacción:

- ✓ **`flush()`**: no confirma la transacción; únicamente **asegura que los cambios programados sean enviados a la *BD***.
- ✓ **`commit()`**: es el encargado de **confirmar definitivamente la transacción**.

Nota: el método `persist()` no realiza cambios inmediatos en la *BD*, sino que simplemente programa un comando para su ejecución posterior.

- ➢ Dependencias (`pom.xml`).

```
<dependency>
```

```
            <groupId>org.mariadb.jdbc</groupId>
            <artifactId>mariadb-java-client</artifactId>
            <version>3.5.1</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.16</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>6.5.2.Final</version>
        </dependency>
```

- Archivo de configuración (`src/main/resources/META-INF/persistence.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
       https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
    <persistence-unit name="pokemon" transaction-type="RESOURCE_LOCAL">
        <!-- Proveedor de JPA (Hibernate) -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- Clases a persistir (AÑADIR RUTA DEL PAQUETE) -->
        <class>Pokemon</class>
        <properties>
            <!-- MariaDB -->
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost/pokemons"/>
            <!-- Credenciales -->
            <property name="jakarta.persistence.jdbc.user" value="root"/>
            <property name="jakarta.persistence.jdbc.password" value="root"/>
            <!-- Automatic schema export -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>
            <!-- SQL statement logging -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Si se desea **que *JPA* cree las tablas en la *BD* cuando inicie la aplicación**, **usar `create`, `drop-and-create` o `update` en la propiedad `jakarta.persistence.schema-generation.database.action`**. Esto garantizará que las tablas se generen según las entidades mapeadas en la aplicación.

- `Pokemon.java`: código del ejemplo del punto 3.2 (se debe tener creada la *BD* `pokemons` y la tabla `pokemon`).
- `PersistirPokemon.java`

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;
import java.math.BigDecimal;

public class PersistirPokemon {
    public static void main(String[] args) {
        // Nombre de la unidad de persistencia (pokemon) debe coincidir con el nombre en persistence.xml
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("pokemon");
                EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            EntityTransaction tx = entityManager.getTransaction();
            tx.begin();
            Pokemon pok = new Pokemon(0,"Bulbasaur", new BigDecimal(6.9), new BigDecimal(0.7), (byte) 45,
                    (byte) 49, (byte) 49, (byte) 65, (byte) 45, Pokemon.Sexo.MH, "Este Pokémon nace con
                    una semilla en el lomo, que brota con el paso del tiempo. Desde que nace, crece
                    alimentándose de los nutrientes que contiene la semilla de su lomo.");
            entityManager.persist(pok);
            tx.commit();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
```

```
}
```

4.3.2 Entidades embebidas.

- Dependencias (`pom.xml`). Ver apartado anterior.
- Archivo de configuración (`src/main/resources/META-INF/persistence.xml`).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
                  https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
    <persistence-unit name="persona" transaction-type="RESOURCE_LOCAL">
        <!-- Proveedor de JPA (Hibernate) -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- Clases a persistir (AÑADIR RUTA DEL PAQUETE) -->
        <class>Persona</class>
        <properties>
            <!-- Configuración de conexión a MariaDB -->
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost/personas"/>
            <property name="jakarta.persistence.jdbc.user" value="root"/>
            <property name="jakarta.persistence.jdbc.password" value="root"/>
            <!-- Generación de esquema -->
            <property name="jakarta.persistence.schema-generation.database.action" value="create"/>
            <!-- Logging de SQL -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

- Crear la *BD* `personas` en el *SGBD MariaDB* (no es necesario crear la tabla).

```sql
CREATE OR REPLACE DATABASE personas COLLATE utf8mb4_spanish_ci;
```

- `Direccion.java`

```java
import jakarta.persistence.Column;
import jakarta.persistence.Embeddable;

@Embeddable
public class Direccion {
    @Column(name = "calle", nullable = false)
    private String calle;
    @Column(name = "ciudad", nullable = false)
    private String ciudad;
    @Column(name = "codigo_postal", nullable = false)
    private String codigoPostal;

    public Direccion() {}
    public void setCalle(String calle) { this.calle = calle; }
    public void setCiudad(String ciudad) { this.ciudad = ciudad; }
    public void setCodigoPostal(String codigoPostal) { this.codigoPostal = codigoPostal; }
}
```

- `Persona.java`

```java
import jakarta.persistence.*;

@Entity
@Table(name = "persona")
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "nombre", nullable = false)
    private String nombre;

    @Embedded
    private Direccion direccion;

    // Constructor sin argumentos (necesario para JPA)
    public Persona() {}
    public Persona(String nombre, Direccion direccion) {
        this.nombre = nombre;
        this.direccion = direccion;
    }
```

```
        // Getters y setters
    }
```

- `PersonasMain.java`

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class PersonasMain {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("persona");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            Direccion direccion = new Direccion();
            direccion.setCalle("Gran vía");
            direccion.setCiudad("Granada");
            direccion.setCodigoPostal("12345");
            Persona persona = new Persona("Juan Moreno", direccion);

            EntityTransaction tx = entityManager.getTransaction();
            tx.begin();
            entityManager.persist(persona);
            tx.commit();

            System.out.println("Persona guardada.");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

4.3.3 Relación uno a uno (1:1).

- Dependencias (`pom.xml`). Ver apartados anteriores.
- `Estudiante.java` (ver código punto 3.3.1 unidireccional).
- `Libro.java` (ver código punto 3.3.1 unidireccional).
- Archivo de configuración (`src/main/resources/META-INF/persistence.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
                  https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
    <persistence-unit name="relacion_11_uni" transaction-type="RESOURCE_LOCAL">
        <!-- Proveedor de JPA (Hibernate) -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <!-- Clases a persistir (AÑADIR RUTA DEL PAQUETE) -->
        <class>Estudiante</class>
        <class>Libro</class>

        <properties>
            <!-- Configuración de conexión a MariaDB -->
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost/estudiantes"/>
            <property name="jakarta.persistence.jdbc.user" value="root"/>
            <property name="jakarta.persistence.jdbc.password" value="root"/>

            <!-- Generación de esquema -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>

            <!-- Logging de SQL -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

- Crear la *BD* `estudiantes` y las tablas necesarias.

```
CREATE OR REPLACE DATABASE estudiantes COLLATE UTF8MB4_SPANISH_CI;
USE estudiantes;
CREATE TABLE libro (
       id INT PRIMARY KEY AUTO_INCREMENT,
       titulo VARCHAR(50)
);
CREATE TABLE estudiante (
       id INT PRIMARY KEY AUTO_INCREMENT,
```

```
            nombre VARCHAR(50),
            id_libro INT,
            FOREIGN KEY (id_libro) REFERENCES libro(id)
        );
```

- `PersistirRelacion11.java`

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class PersistirRelacion11 {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("relacion_11_uni");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            Libro libro = new Libro("Historia del Arte");
            Estudiante estudiante = new Estudiante("Juan", libro);

            EntityTransaction tx = entityManager.getTransaction();
            tx.begin();
            entityManager.persist(estudiante);
            tx.commit();

            System.out.println("Estudiante y libro persistidos correctamente.");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

El **código para persistir entidades en una relación bidireccional es similar al unidireccional**. Es muy similar porque *JPA* se encarga de resolver las relaciones definidas por las anotaciones.

4.3.4 Relación uno a muchos (1:N).

- Dependencias (`pom.xml`). Ver apartados anteriores.
- `Autor.java` (ver código punto 3.3.2). Agregar el siguiente método:

```
public List<Libro> getLibros() { return libros; }
```

- `Libro.java` (ver código punto 3.3.2).
- Archivo de configuración (`src/main/resources/META-INF/persistence.xml`). Modificar:

```
<!-- Configuración de conexión a MariaDB -->
<property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost/autores_libros"/>

<!-- Clases a persistir (AÑADIR RUTA DEL PAQUETE) -->
<class>Autor</class>
<class>Libro</class>
```

- Crear la *BD* `autores_libros` y las tablas necesarias.

```
CREATE OR REPLACE DATABASE autores_libros COLLATE utf8mb4_spanish_ci;
USE autores_libros;
CREATE TABLE autor (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre_autor VARCHAR(50)
);
CREATE TABLE libro (
    id INT PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(50),
    id_autor INT,
    FOREIGN KEY (id_autor) REFERENCES autor(id)
);
```

- `PersistirRelacion1N.java` → Colocar dentro del bloque `try` el siguiente código:

```
Autor autor = new Autor("Juan Moreno");
Libro libro1 = new Libro("Programación", autor);
Libro libro2 = new Libro("Bases de Datos", autor);

autor.getLibros().add(libro1);
autor.getLibros().add(libro2);

EntityTransaction tx = entityManager.getTransaction();
tx.begin();
entityManager.persist(autor);
tx.commit();
```

```
System.out.println("Autor y libros asociados persistidos correctamente.");
```

4.3.5 Relación muchos a muchos (N:M).

- Dependencias (`pom.xml`). Ver apartados anteriores.
- `Estudiante.java` (ver código punto 3.3.3). Agregar el siguiente método:

```
public List<Curso> getCursos() { return cursos; }
```

- `Curso.java` (ver código punto 3.3.3). Agregar el siguiente método:

```
public List<Estudiante> getEstudiantes() { return estudiantes; }
```

- Archivo de configuración (`src/main/resources/META-INF/persistence.xml`). Modificar:

```
<!-- Configuración de conexión a MariaDB -->
<property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost/estudiantes2"/>

<!-- Clases a persistir (AÑADIR RUTA DEL PAQUETE) -->
<class>Estudiante</class>
<class>Curso</class>
```

- Crear la *BD* `estudiantes2` y las tablas necesarias.

```
CREATE OR REPLACE DATABASE estudiantes2 COLLATE utf8mb4_spanish_ci;
USE estudiantes2;
CREATE TABLE estudiante (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50)
);
CREATE TABLE curso (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(50)
);
CREATE TABLE cursa (
    id_estudiante INT,
    id_curso INT,
    PRIMARY KEY (id_estudiante, id_curso),
    FOREIGN KEY (id_estudiante) REFERENCES estudiante(id),
    FOREIGN KEY (id_curso) REFERENCES curso(id)
);
```

- `PersistirRelacionNM.java` → Colocar dentro del bloque `try` el siguiente código:

```
Estudiante estudiante1 = new Estudiante("Juan Moreno");
Estudiante estudiante2 = new Estudiante("María Ruiz");
Curso curso1 = new Curso("Matemáticas");
Curso curso2 = new Curso("Historia");

estudiante1.getCursos().add(curso1);
estudiante2.getCursos().add(curso1);
estudiante2.getCursos().add(curso2);
curso1.getEstudiantes().add(estudiante1);
curso1.getEstudiantes().add(estudiante2);
curso2.getEstudiantes().add(estudiante2);

EntityTransaction tx = entityManager.getTransaction();
tx.begin();
entityManager.persist(estudiante1);
entityManager.persist(estudiante2);
entityManager.persist(curso1);
entityManager.persist(curso2);
tx.commit();

System.out.println("Estudiante, cursos y asociaciones persistidos correctamente.");
```

TAREA

4. **Crear una aplicación que permita al usuario introducir por teclado la información de un coche (matrícula, marca, modelo y número de plazas) y guarde dichos datos en la *BD* utilizando *JPA*.**
5. **Crear una aplicación que, basándose en las entidades y la unidad de persistencia creadas en el ejercicio 2, solicite al usuario los valores necesarios por teclado y los almacene en la *BD* mediante *JPA*.**
6. **Crear una aplicación que, utilizando las entidades y la unidad de persistencia definidas en el ejercicio 3, permita al usuario introducir los datos requeridos por teclado y los registre en la *BD* haciendo uso de *JPA*.**

4.4 Carga de objetos.

El **método `find()`** de `EntityManager` **permite recuperar una instancia persistente de una entidad, utilizando el identificador del registro**. Este método **toma como parámetros la clase de la entidad y la clave primaria que la identifica. Devuelve la instancia de la entidad encontrada, o `null` si no se encuentra ninguna coincidencia**. Cuando se llama a `find()`, el **objeto devuelto se maneja como una entidad, lo que significa que se integra en el contexto de persistencia actual**.

Existen **otras formas de cargar objetos** con *JPA*, que se explorarán en el siguiente apartado de consultas:

- ✓ ***JPQL***.
- ✓ ***Criteria Queries*** (consultas programáticas).
- ✓ **Consultas nativas *SQL***.
- ✓ **Consultas con nombre**.

> **`getReference()` y `find()` son similares**, pero `find()` carga inmediatamente los datos completos de la entidad, mientras que **`getReference()` devuelve una referencia y carga los datos solo cuando se acceden**.

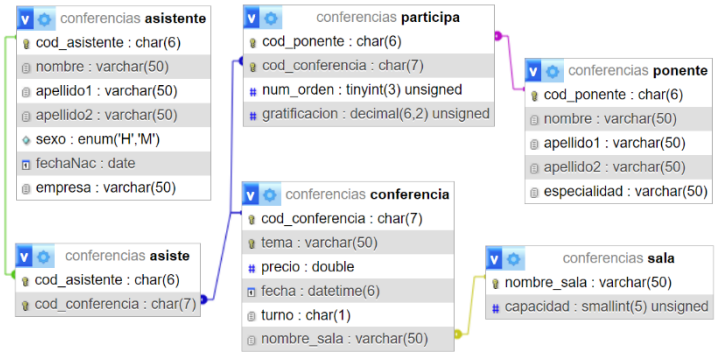
En el siguiente apartado de consultas, se presentará un ejemplo completo de uso del método `find()`. A continuación, se muestra una parte del código:

Ejemplo:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("nombre_unidad_persistencia");
EntityManager em = emf.createEntityManager();
Integer idPersona = 1; // Suponer que se quiere cargar la persona con ID 1
Persona persona = em.find(Persona.class, idPersona);
if (persona != null) {
    System.out.println("Persona encontrada:");
    System.out.println("Nombre: " + persona.getNombre());
    System.out.println("Edad: " + persona.getEdad());
} else
    System.out.println("Persona no encontrada con ID " + idPersona);
em.close();
emf.close();
```

4.5 Consultas.

Los ejemplos de este apartado se basarán en la *BD* `conferencias`. El script para su creación está disponible para su descarga en *Moodle Centros*.



4.5.1 Consultas con lenguaje específico del ORM.

JPA permite realizar consultas a la *BD* utilizando ***JPQL*** (*Java/Jakarta Persistence Query Language*), un **lenguaje orientado a objetos basado en *SQL***. *JPA* se encarga de traducir las sentencias *JPQL* a *SQL* para que sean ejecutadas.

Características de *JPQL*:

- **Sintaxis similar a *SQL***: aunque se asemeje a *SQL*, **en lugar de trabajar con tablas y columnas**, *JPQL* **opera con entidades, sus atributos y sus relaciones**.
- **Consulta en objetos**: las **consultas se realizan sobre los objetos del modelo de dominio**, no directamente sobre la *BD* subyacente.
- **Independencia del *SGBD***: *JPQL* es independiente del *SGBD* subyacente, lo que facilita la portabilidad entre diferentes proveedores. Mientras que *SQL* tiene particularidades según el fabricante, *JPQL* mantiene su consistencia en diferentes plataformas y proveedores.
- **Cláusula `SELECT`**: se utiliza para **seleccionar atributos o la entidad completa** que se desea recuperar. Ejemplo: `SELECT e.nombre, e.edad FROM MiEntidad e`
- **Parámetros y enlace de parámetros**: permite utilizar parámetros en las consultas y enlazar valores a estos parámetros. Ejemplo: `SELECT e FROM MiEntidad e WHERE e.nombre = :nombreParam`
- **Funciones de agregación**: se pueden usar funciones como **`AVG`**, **`SUM`**, **`COUNT`**, **`MAX`**, **`MIN`**, etc., para realizar cálculos dentro de las consultas. Ejemplo: `SELECT AVG(e.edad) FROM MiEntidad e`
- **Subconsultas**: *JPQL* permite incluir subconsultas dentro de consultas principales para realizar operaciones más complejas. Ejemplo: `SELECT e FROM MiEntidad e WHERE e.valor = (SELECT MAX(v) FROM OtraEntidad)`

- **Ordenamiento**: se puede ordenar los resultados mediante la cláusula **`ORDER BY`**. Ejemplo: `SELECT e FROM MiEntidad e ORDER BY e.nombre ASC`

Resumen **básico** de cláusulas en *JPQL*:

- Cláusula **`FROM`**: **define la entidad sobre la cual se realizará la consulta**.
- Cláusula **`WHERE`**: **filtra los resultados según condiciones específicas**.
- Cláusula **`JOIN`**: permite **realizar uniones entre entidades relacionadas**.

Desventajas de *JPQL*:
- Limitación de expresividad.
- Rendimiento.
- Depuración y trazabilidad.
- Curva de aprendizaje.

Para **crear y ejecutar consultas** *JPQL*, se utilizan los métodos disponibles en *JPA*. El **método `createQuery` devuelve un objeto que implementa la interfaz `Query`**. Esta interfaz **ofrece métodos para ejecutar la consulta y obtener los resultados**, gestionando así el proceso de consulta.

`EntityManager` **method**	`Query` **execution method**
`createQuery(String,Class)`	`getResultList()`, `getSingleResult()`, `getSingleResultOrNull()`

Los **métodos más utilizados** de las **interfaces `Query`** y **`TypedQuery`** son:

Método	Descripción
`List<T> getResultList()`	Devuelve una lista de resultados de la consulta.
`T getSingleResult()`	Devuelve un único resultado de la consulta. Puede lanzar una `NoResultException` o `NonUniqueResultException` si no se encuentra un único resultado.
`T o null getSingleResultOrNull()`	Devuelve un único resultado de la consulta o `null` si no se encuentran resultados.
`List<T> list()`	Obtiene los resultados de la consulta como una lista.
`Query setParameter(String, Object)`	Asigna un valor a un parámetro de la consulta utilizando el nombre del parámetro.
`Query setParameter(int, Object)`	Asigna un valor a un parámetro de la consulta utilizando un índice.
`Query setMaxResults(int)`	Establece el número máximo de resultados a recuperar de la consulta (limit).
`Query setFirstResult(int)`	Establece el índice del primer resultado a recuperar (offset).
`Query setFetchSize(int)`	Establece el tamaño del conjunto de resultados que se recuperará para la consulta.
`int executeUpdate()`	Ejecuta la consulta para operaciones de actualización, eliminación, etc. Devuelve el número de registros afectados. *** **Exclusivo de Query** ***

Si se espera que una consulta devuelva un **único resultado**, se pueden utilizar los métodos:

- ✓ **`getSingleResult()` lanza una excepción** (`NoResultException` o `NonUniqueResultException`) **si no se encuentra ningún resultado o si se encuentran múltiples resultados**.
- ✓ **`getSingleResultOrNull()` devuelve `null` si no se encuentran resultados**, sin lanzar ninguna excepción.

La forma de ejecutar una consulta haciendo uso de `Query` es la siguiente:

```
// Crear la consulta utilizando JPQL
String jpql = "SELECT u FROM Usuario u WHERE u.nombre = :nombreUsuario";
Query query = entityManager.createQuery(jpql); // También se puede usar TypedQuery<Usuario>
// TypedQuery<Usuario> query = entityManager.createQuery(jpql, Usuario.class);
query.setParameter("nombreUsuario", "Juan");
List<Usuario> usuarios = query.getResultList(); // Obtener el resultado de la consulta
// Realizar operaciones con los resultados (en este caso, imprimir los nombres)
for (Usuario usuario : usuarios)
    System.out.println("Nombre: " + usuario.getNombre());
```

El **`SELECT`** en *JPQL* **permite seleccionar atributos específicos de las entidades o aplicar funciones de agregación**. Dependiendo de los resultados de la consulta, la **ejecución devolverá diferentes tipos de objetos**:

- ✓ **`List<Object[]>`**: cuando **la consulta devuelve múltiples columnas o propiedades por fila, pero no se necesitan los resultados como objetos completos de entidades**. Cada elemento en la lista es un array de objetos, donde cada elemento del array representa una columna o propiedad devuelta por la consulta.

```
String jpql = "SELECT e.id, e.nombre FROM Empleado e";
Query query = entityManager.createQuery(jpql);
List<Object[]> results = query.getResultList();

for (Object[] result : results) {
    Integer id = (Integer) result[0];
    String nombre = (String) result[1];

    // Hacer algo con id y nombre
}
```

En este caso se puede usar un enfoque más seguro y claro creando una clase personalizada (DTO) para encapsular los resultados de la consulta, lo que evita manejar un array de objetos genéricos (`Object[]`). Esto mejora la legibilidad y la seguridad del tipo. Se comenta más adelante en este mismo punto.

- ✓ **`List<Object>`**: cuando la **consulta devuelve un solo valor por fila**. Cada elemento en la lista es un objeto que representa una fila de la consulta.

```
String jpql = "SELECT e.nombre FROM Empleado e";
Query query = entityManager.createQuery(jpql);
List<Object> resultados = query.getResultList();

for (Object resultado : resultados)
    System.out.println("Nombre: " + resultado);
```

```
TypedQuery<String> query = entityManager.createQuery(jpql, String.class);
List<String> nombres = query.getResultList();
```

- ✓ **`Object`**: cuando la **consulta devuelve un único valor y una única fila** (como en una consulta de agregación), se puede **utilizar `getSingleResult()`** que **devuelve un `Object`** con el resultado.

```
String jpqlCount = "SELECT COUNT(e) FROM Empleado e";
Query queryCount = entityManager.createQuery(jpqlCount);
Object resultadoCount = queryCount.getSingleResult();

System.out.println("Resultado de COUNT: " + resultadoCount); // Mostrar resultado de COUNT
```

```
TypedQuery<Long> queryCount =
    entityManager.createQuery(jpqlCount, Long.class);
Long resultadoCount = queryCount.getSingleResult();
```

En ocasiones, **es útil parametrizar una consulta para obtener resultados diferentes según los valores asignados a los parámetros.** A esto se les conoce como **consultas dinámicas**. Los parámetros de una consulta **pueden ser**:

- ✓ **Posicionales**: **se denotan con el símbolo de cierre de interrogación (?) seguido del número del parámetro** (?1, ?2, ?3, …).
- ✓ **Nominales**: **se denotan con el nombre asignado precedido de dos puntos** (:).

Los **parámetros solo se pueden usar en las cláusulas `WHERE` o `HAVING`**, y **no se pueden mezclar** parámetros posicionales y nominales en la misma consulta.

Antes de ejecutar estas sentencias es preciso **asignar valores a los parámetros**, para lo cual se dispone del **método `setParameter()`** de la **clase `Query`/`TypedQuery`**:

```
String jpql = "SELECT e FROM Empleado e WHERE e.departamento = :nombreDepartamento";
TypedQuery<Empleado> query = em.createQuery(jpql, Empleado.class);
query.setParameter("nombreDepartamento", "Ventas");
List<Empleado> empleados = query.getResultList();
```

Ejemplo: consultas sobre las tablas `sala` y `conferencia` de la *BD* `conferencias` haciendo uso de *JPQL*.

- ➢ Dependencias (`pom.xml`). Ver apartados anteriores.
- ➢ `Sala.java` → Se debe tener en cuenta al realizar el mapeo la relación 1:N entre `sala` y `conferencia`.

```
import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "sala")
public class Sala {
    @Id
    @Column(name = "nombre_sala")
    private String nombreSala;
    @Column(name = "capacidad")
    private int capacidad;
    @OneToMany(mappedBy = "nombreSala", cascade = CascadeType.ALL)
    private List<Conferencia> conferencias = new ArrayList<>();
    @Override
    public String toString() {
        return "Sala { " + "nombreSala = '" + nombreSala + "' , capacidad = " + capacidad + " }";
    }
}
```

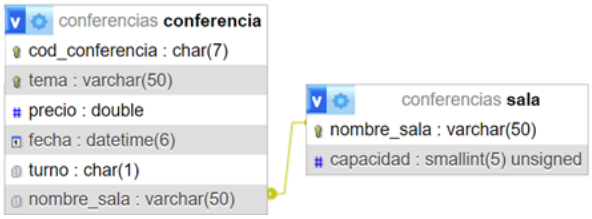
> Query es una interfaz general para consultas sin tipo específico de resultado, mientras que TypedQuery es más segura, ya que permite trabajar con un tipo conocido de resultado sin necesidad de conversiones. Se recomienda usar TypedQuery cuando se conoce el tipo de los resultados, ya que es más limpio y seguro.
>
> `Query query = entityManager.createQuery("SELECT e FROM Empleado e");`
>
> `TypedQuery<Empleado> query = entityManager.createQuery("SELECT e FROM Empleado e", Empleado.class);`

- ➢ `Conferencia.java` → Se debe tener en cuenta al realizar el mapeo la relación 1:N entre `sala` y `conferencia`

```
import jakarta.persistence.*;
import java.math.BigDecimal;
import java.text.SimpleDateFormat;
import java.util.Date;

@Entity
@Table(name = "conferencia")
public class Conferencia {
    @Id
    @Column(name = "cod_conferencia")
    private String codConferencia;
    @Column(name = "tema")
    private String tema;
    @Column(name = "precio")
    private BigDecimal precio;
```

> La **opción *Run JPQL Query* en *NetBeans*** es una **característica útil para probar consultas *JPQL* directamente** desde el entorno de desarrollo. Abrir la consola de consultas *JPQL*:
>
> 1. Hacer clic derecho en el archivo `persistence.xml` en el explorador de proyectos.
> 2. Seleccionar *Run JPQL Query* en el menú contextual.
>
> Nota: ***Run JPQL Query* en *NetBeans* está diseñado para interactuar específicamente con *EclipseLink***. Algunas funcionalidades dependen de características internas de *EclipseLink* que no están disponibles en *Hibernate*.

```
        @Column(name = "fecha")
        private Date fecha;
        @Column(name = "turno")
        private char turno;
        @ManyToOne
        @JoinColumn(name = "nombre_sala")
        private Sala nombreSala;
        @Override
        public String toString() {
            SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
            return "Conferencia { codConferencia = '" + codConferencia + "', tema = '" + tema +
                    "', precio = " + precio + ", fecha = '" + sdf.format(fecha) + "', turno = '" + turno +
                    "', nombreSala = '" + nombreSala + "' }";
        }
    }
```



- Archivo de configuración (`src/main/resources/META-INF/persistence.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
       https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
    <persistence-unit name="conferencias">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!-- Clases a persistir -->
        <class>Sala</class>
        <class>Conferencia</class>
        <properties>
            <!-- Configuración de conexión a MariaDB -->
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost/conferencias"/>
            <property name="jakarta.persistence.jdbc.user" value="conferencias"/>
            <property name="jakarta.persistence.jdbc.password" value="conferencias"/>
            <!-- Generación de esquema -->
            <property name="jakarta.persistence.schema-generation.database.action" value="none"/>
            <!-- Logging de SQL -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

- ConsultasConferenciasJPQL.java

> Se puede utilizar la clase `Object[]` para recuperar datos de una consulta en la que intervienen varias tablas (o campos específicos de una) y no se tiene una clase específica que represente exactamente los resultados de la consulta (ver ejemplos).
>
> También se pueden recuperar los valores de una consulta con un array de objetos (`List<Object[]>`).

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import jakarta.persistence.Query;
import jakarta.persistence.TypedQuery;
import java.util.List;
public class ConsultasConferenciasJPQL {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("conferencias");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
// 1. Consultar conferencias por id
            Conferencia conf1 = entityManager.find(Conferencia.class, "POO1314");
            System.out.println("(1) Datos de la conferencia 'POO1314': ");
            System.out.println(conf1);
// 2. Consultar todas las conferencias
    TypedQuery<Conferencia> queryCons2 = entityManager.createQuery("SELECT c FROM Conferencia c", Conferencia.class);
    List<Conferencia> conf2 = queryCons2.getResultList();
    System.out.println("(2) Todas las conferencias de la tabla: ");
    for (Conferencia conferencia : conf2)
        System.out.println(conferencia);
// 3. Consultar las conferencias que se dan en la sala 'Afrodita'
    TypedQuery<Conferencia> queryCons3 = entityManager.createQuery("SELECT c FROM Conferencia c WHERE
                    c.nombreSala.nombreSala = :nombreSala", Conferencia.class);
    queryCons3.setParameter("nombreSala", "Afrodita");
    List<Conferencia> conf3 = queryCons3.getResultList();
    System.out.println("(3) Conferencias que se dan en la sala 'Afrodita': ");
    for (Conferencia conferencia : conf3)
        System.out.println(conferencia);
// 4. Obtener el precio medio de las conferencias del turno de mañana
            Query queryCons4 = entityManager.createQuery("SELECT AVG(c.precio) FROM Conferencia c WHERE c.turno = ?1");
            queryCons4.setParameter(1, 'M');
            double media = (double) queryCons4.getSingleResult();
```

```
            System.out.print("(4) Precio medio de las conferencias del turno de 'M': ");
            System.out.println(media);
// 5. Obtener el código y el tema de las conferencias cuyo precio esté comprendido entre 12 y 19, ordenado por el tema de la conferencia
        Query queryCons5 = entityManager.createQuery("SELECT c.codConferencia, c.tema FROM Conferencia c WHERE c.precio
                    BETWEEN :min AND :max ORDER BY c.tema");  // Se podría usar DTO TypedQuery<ConferenciaDTO>
            queryCons5.setParameter("min", 12);
            queryCons5.setParameter("max", 19);
            List<Object[]> conf5 = queryCons5.getResultList();
            System.out.println("(5) Cód. y tema de las conferencias entre 12 y 19, ordenadas por tema de forma ascendente: ");
            for (Object[] conferencia : conf5)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
// 6. Obtener el código y el tema de las conferencias cuyo precio esté comprendido entre 12 y 19, y se den en la sala 'Afrodita', ordenado por el tema de la conferencia
        Query queryCons6 = entityManager.createQuery("SELECT c.codConferencia, c.tema FROM Conferencia c WHERE c.precio
                    BETWEEN :min AND :max AND c.nombreSala.nombreSala = :sala ORDER BY c.tema");
            queryCons6.setParameter("min", 12);
            queryCons6.setParameter("max", 19);
            queryCons6.setParameter("sala", "Afrodita");
            List<Object[]> conf6 = queryCons6.getResultList();
            System.out.println("(6) Cód. y tema de las conferencias entre 12 y 19, y que se dan en la sala 'Afrodita' ordenadas por tema de forma ascendente: ");
            for (Object[] conferencia : conf6)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
// 7. JOIN - Obtener el código, el tema de la conferencia y la capacidad de la sala para aquellas cuyo precio esté comprendido entre 12 y 19
      Query queryCons7 = entityManager.createQuery("SELECT c.codConferencia, c.tema, s.capacidad FROM Conferencia c
                    JOIN Sala s ON c.nombreSala = s WHERE c.precio BETWEEN :min AND :max");
            queryCons7.setParameter("min", 12);
            queryCons7.setParameter("max", 19);
            List<Object[]> conf7 = queryCons7.getResultList();
            System.out.println("(7) Cód., tema y capacidad de la sala de las conferencias cuyo precio está comprendido entre 12 y 19: ");
            for (Object[] conferencia : conf7)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1] + " : " + conferencia[2]);
// 8. SUBCONSULTAS - Obtener el código y el tema de las conferencias que se dan en una sala con capacidad >= 200
       Query queryCons8 = entityManager.createQuery("SELECT c.codConferencia, c.tema FROM Conferencia c WHERE
                    c.nombreSala IN (SELECT s FROM Sala s WHERE s.capacidad >= :capacidad)");
       queryCons8.setParameter("capacidad", 200);
       List<Object[]> conf8 = queryCons8.getResultList();
       System.out.println("(8) Cód. y tema de las conferencias que se dan en una sala con una capacidad >= 200: ");
       for (Object[] conferencia : conf8)
           System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
        }
    }
}
```

Alternativas al uso de `List<Object[]>` en consultas con *JPQL*, que hacen el código más legible y seguro:

- ✓ **Uso de *DTOs*** (Data Transfer Objects): usar una clase específica para encapsular los datos y configurar la consulta para que devuelva instancias de esta clase.

```
package paquete; // Cambiar "paquete" por el paquete correspondiente
public class ConferenciaDTO {
    private int codConferencia;
    private String tema;

    // Constructor para inicializar los atributos.
    public ConferenciaDTO(int codConferencia, String tema) {
        this.codConferencia = codConferencia;
        this.tema = tema;
    }
    public int getCodConferencia() { return codConferencia; }
    public void setCodConferencia(int codConferencia) { this.codConferencia = codConferencia; }
    public String getTema() { return tema; }
    public void setTema(String tema) { this.tema = tema; }
    @Override
    public String toString() { return codConferencia + " : " + tema; }
}
```

```
// ...
TypedQuery<ConferenciaDTO> query = entityManager.createQuery("SELECT new
                                        paquete.ConferenciaDTO(c.codConferencia, c.tema) " +
    "FROM Conferencia c WHERE c.precio BETWEEN :min AND :max ORDER BY c.tema", ConferenciaDTO.class);
```

```
query.setParameter("min", 12);
query.setParameter("max", 19);
List<ConferenciaDTO> resultados = query.getResultList();
// ...
```

- ✓ **Mapeo a entidades**: seleccionar entidades completas en lugar de atributos individuales.

```
// ...
TypedQuery<Conferencia> query = entityManager.createQuery("SELECT c FROM Conferencia c WHERE" +
    " c.precio BETWEEN :min AND :max ORDER BY c.tema", Conferencia.class);
query.setParameter("min", 12);
query.setParameter("max", 19);
List<Conferencia> conferencias = query.getResultList();
for (Conferencia conferencia : conferencias)
    System.out.println(" - " + conferencia.getCodConferencia() + " : " + conferencia.getTema());
// ...
```

- ✓ **Uso de `Tuple`**: proporciona acceso basado en nombres o índices.

```
// ...
TypedQuery<Tuple> query = entityManager.createQuery("SELECT c.codConferencia AS cod, c.tema AS tema" +
   " FROM Conferencia c WHERE c.precio BETWEEN :min AND :max ORDER BY c.tema", Tuple.class);
query.setParameter("min", 12);
query.setParameter("max", 19);
List<Tuple> resultados = query.getResultList();
for (Tuple resultado : resultados)
    System.out.println(" - " + resultado.get("cod") + " : " + resultado.get("tema"));
// ...
```

- ✓ **Clases anónimas o registros** (*Java 16+*): estructura ligera similar a un *DTO*.

```
public record ConferenciaRecord(int codConferencia, String tema) {}

// ...
TypedQuery<ConferenciaRecord> query = entityManager.createQuery("SELECT new
                                        paquete.ConferenciaRecord(c.codConferencia, c.tema) " +
    "FROM Conferencia c WHERE c.precio BETWEEN :min AND :max ORDER BY c.tema", ConferenciaRecord.class);
query.setParameter("min", 12);
query.setParameter("max", 19);
List<ConferenciaRecord> resultados = query.getResultList();
// ...
```

Para **evitar el uso de `List<Object>`** se puede trabajar con tipos específicos:

```
// ...
String jpql = "SELECT e.nombre FROM Empleado e";
TypedQuery<String> query = entityManager.createQuery(jpql, String.class);
List<String> nombres = query.getResultList();

for (String nombre : nombres)
    System.out.println("Nombre: " + nombre);
// ...
```

Ejemplo: más consultas sobre las tablas `sala` y `conferencia` de la *BD* `conferencias` haciendo uso de *JPQL*.

```
public record SalaConferencia (String nombConf, int capacidad) {}
-----------------------------------------------------------------------------------------------------------------------
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import java.util.List;

public class ConsultasConferenciasJPQL2 {
    public static void main(String[] args) {
        try (EntityManagerFactory emf = Persistence.createEntityManagerFactory("conferencias");
                EntityManager em = emf.createEntityManager()) {
// 1. Consultar las conferencias cuyo tema contiene la palabra "Programación"
            List<Conferencia> c1 = em.createQuery("SELECT c FROM Conferencia c "
                    + "WHERE c.tema LIKE :confTema ORDER BY c.tema", Conferencia.class)
                    .setParameter("confTema", "%programación%")
// List<Conferencia> c1 = em.createQuery("SELECT c FROM Conferencia c WHERE c.tema LIKE ?1", Conferencia.class)
//         .setParameter(1, "%programación%")
                    .setMaxResults(10)
                    .getResultList();
            System.out.println("(1) Conferencias que incluyen el texto 'programación' en su tema: ");
            for (Conferencia conferencia : c1)
                System.out.println(" - " + conferencia);

// 2. Consultar las distintas salas del turno de tarde donde se celebran conferencias
            List<String> c2 = em.createQuery("SELECT DISTINCT c.nombreSala.nombreSala FROM Conferencia c "
                    + "WHERE c.turno = :confTurno ORDER BY 1")
                    .setParameter("confTurno", 'T')
```

```
                .getResultList();
        System.out.println("(2) Salas distintas del truno de tarde donde se celebran conferencias: ");
        for (String sala : c2)
            System.out.println(" - " + sala);
// 3. Consultar las conferencias que se dan en el turno de tarde y su precio es superior a 15
        List<Conferencia> c3 = em.createQuery("SELECT c FROM Conferencia c WHERE c.turno = :confTurno "
                + "AND c.precio > :confPrecio ORDER BY c.tema", Conferencia.class)
                .setParameter("confTurno", 'T')
                .setParameter("confPrecio", 15)
                .getResultList();
        System.out.println("(3) Conferencias que se dan en el turno de tarde y su precio es superior a 15: ");
        for (Conferencia conferencia : c3)
            System.out.println(" - " + conferencia);
// 4. Consultar las conferencias que se dan en las salas con capacidad superior a 150
        List<Conferencia> c4 = em.createQuery("SELECT c FROM Conferencia c JOIN c.nombreSala s "
                + "WHERE s.capacidad > :salaCapacidad ORDER BY c.tema", Conferencia.class)
                .setParameter("salaCapacidad", 150)
                .getResultList();
        System.out.println("(4) Conferencias que se dan en las salas con capacidad superior a 150: ");
        for (Conferencia conferencia : c4)
            System.out.println(" - " + conferencia);
// 5. Obtener las salas donde no se da ninguna conferencia
        List<Sala> c5 = em.createQuery("SELECT DISTINCT s FROM Sala s WHERE s.conferencias IS EMPTY "
                + "ORDER BY s.nombreSala", Sala.class)
                .getResultList();
        System.out.println("(5) Salas distintas del turno de tarde donde se celebran conferencias: ");
        for (Sala sala : c5)
            System.out.println(" - " + sala);
// 6. Obtener las salas donde se celebran conferencias cuyo precio es NULL
        List<Sala> c6 = em.createQuery("SELECT DISTINCT s FROM Sala s JOIN Conferencia c ON s = "
                + "c.nombreSala WHERE c.precio IS NULL ORDER BY s.nombreSala", Sala.class)
                .getResultList();
        System.out.println("(6) Salas donde se celebran conferencias cuyo precio es NULL: ");
        for (Sala sala : c6)
            System.out.println(" - " + sala);
// 7. Obtener las salas en las que se dan conferencias cuyo precio está comprendido entre 12 y 19
        List<Sala> c7 = em.createQuery("SELECT DISTINCT s FROM Sala s JOIN Conferencia c ON s = "
             + "c.nombreSala WHERE c.precio BETWEEN :precMin AND :precMax ORDER BY s.nombreSala", Sala.class)
   // + "ON s = c.nombreSala WHERE c.precio >= :precMin AND c.precio <= :precMax", Sala.class)
                .setParameter("precMin", 12)
                .setParameter("precMax", 19)
                .getResultList();
        System.out.println("(7) Salas en las que se dan conferencias cuyo precio está comprendido entre 12 y 19: ");
        for (Sala sala : c7)
            System.out.println(" - " + sala);
// 8. Obtener las salas cuya capacidad es mayor que la de 'Hermes'
        List<Sala> c8 = em.createQuery("SELECT DISTINCT s1 FROM Sala s1, Sala s2 "
                + "WHERE s1.capacidad > s2.capacidad AND s2.nombreSala = :nomSala ORDER BY s1.nombreSala", Sala.class)
  // + "ON s.nombreSala = c.nombreSala.nombreSala WHERE c.precio >= :precMin AND c.precio <= :precMax", Sala.class)
                .setParameter("nomSala", "Hermes")
                .getResultList();
        System.out.println("(8) Salas cuya capacidad es mayor que la de 'Hermes': ");
        for (Sala sala : c8)
            System.out.println(" - " + sala);
// 9. Mostrar todas las salas junto a las conferencias que se dan en ellas (inclida Hermes)
       List<Object[]> c9 = em.createQuery("SELECT s, c FROM Sala s LEFT JOIN s.conferencias c ORDER BY s.nombreSala, c.tema")
                .getResultList();
        System.out.println("(9) Salas junto a las conferencias que se dan en ellas (incluida Hermes): ");
        for (Object[] o : c9)
            System.out.println(" - " + o[0] + " : " + o[1]);
// 10. Conferencias que se dan en las salas Afrodita y Zeus con IN
        List<Conferencia> c10 = em.createQuery("SELECT c FROM Conferencia c WHERE c.nombreSala.nombreSala IN (:s1, :s2) "
                + "ORDER BY c.tema", Conferencia.class).setParameter("s1", "Afrodita")
                .setParameter("s2", "Zeus")
                .getResultList();
        System.out.println("(10) Conferencias que se dan en las salas Afrodita y Zeus con IN: ");
        for (Conferencia c : c10)
            System.out.println(" - " + c);
// 11. Sala en la que se da la conferencia ADS1314 con MEMBER OF (miembro de una colección)
        Conferencia conf = em.find(Conferencia.class, "ADS1314");
        List<Sala> c11 = em.createQuery("SELECT s FROM Sala s WHERE :conf MEMBER OF s.conferencias", Sala.class)
                .setParameter("conf", conf)
                .getResultList();
        System.out.println("(11) Sala en la que se da la conferencia ADS1314 con MEMBER OF (miembro de una colección): ");
        for (Sala s : c11)
            System.out.println(" - " + s);
// 12. Obtener las salas en las que se han dado dos o más conferencias (con subconsultas)
  List<Sala> c12 = em.createQuery("SELECT s FROM Sala s WHERE (SELECT COUNT(c) FROM s.conferencias c) >= :nConf", Sala.class)
                .setParameter("nConf", 2)
```

```
                .getResultList();
        System.out.println("(12) Salas en las que se han dado dos o más conferencias (con subconsultas): ");
        for (Sala s : c12)
            System.out.println(" - " + s);
// 13. Obtener las salas donde no se da ninguna conferencia con NOT EXISTS y MEMBER OF
        List<Sala> c13 = em.createQuery("SELECT DISTINCT s FROM Sala s WHERE NOT EXISTS (SELECT c FROM Conferencia c "
                + "WHERE c MEMBER OF s.conferencias) ORDER BY s.nombreSala", Sala.class)
                .getResultList();
        System.out.println("(13) Salas donde no se da ninguna conferencia con NOT EXISTS y MEMBER OF: ");
        for (Sala sala : c13)
            System.out.println(" - " + sala);
// 14. Obtener las conferencias con precio mayor al de cualquier conferencia del turno de tarde (T)
        List<Conferencia> c14 = em.createQuery("SELECT c FROM Conferencia c WHERE c.precio > ALL (SELECT c.precio FROM "
                + "Conferencia c WHERE c.turno = :turnoConf) ORDER BY c.tema", Conferencia.class)
                .setParameter("turnoConf", 'T')
                .getResultList();
        System.out.println("(14) Conferencias con precio mayor al de cualquier conferencia del turno de tarde (>ALL): ");
        for (Conferencia conferencia : c14)
            System.out.println(" - " + conferencia);
// 15. Clasificar las conferencias por precio con CASE - WHEN
        List<Object[]> c15 = em.createQuery("SELECT c.codConferencia, c.tema, c.turno, "
                                            + "CASE "
                                            + "    WHEN c.precio >= 20 THEN 'Alto' "
                                            + "    WHEN c.precio >= 15 THEN 'Medio' "
                                            + "    WHEN c.precio >= 10 THEN 'Bajo' "
                                            + "    ELSE 'Actualizar precio' "
                                            + "END "
                                            + "FROM Conferencia c ORDER BY c.tema")
                .getResultList();
        System.out.println("(15) Clasificar las conferencias por precio con CASE - WHEN: ");
        for (Object[] o : c15)
            System.out.println(" - { " + o[0] + " : " + o[1] + " : " + o[2] + " : " + o[3] + " }");
// 16. Obtener total de conferencias, media de precios, precio máximo y precio mínimo por turno
        List<Object[]> c16 = em.createQuery("SELECT c.turno, COUNT(c), AVG(c.precio), MAX(c.precio), MIN(c.precio), "
                + " SUM(c.precio) FROM Conferencia c GROUP BY c.turno")
                .getResultList();
        System.out.println("(16) Total de conferencias, media de precios, precio máximo y precio mínimo por turno: ");
        for (Object[] o : c16)
            System.out.println(" - Turno: " + o[0] + ", total: " + o[1] + ", media: " + o[2] + ", máximo: " + o[3]
                    + ", mínimo: " + o[4] + ", sumna precios: " + o[5] + ".");
// 17. Obtener el turno cuyo precio medio de las conferencias de ese turno es superior a 15
        List<Object[]> c17 = em.createQuery("SELECT c.turno, AVG(c.precio) FROM Conferencia c GROUP BY c.turno "
                + "HAVING AVG(c.precio) > :precio")
                .setParameter("precio", 15)
                .getResultList();
        System.out.println("(17) Total de conferencias, media de precios, precio máximo y precio mínimo por turno: ");
        for (Object[] o : c17)
            System.out.println(" - Turno: " + o[0] + ".");
// 18. Obtener el nombre de la conferencia junto a la capacidad de la sala en la que se da (record)
        List<SalaConferencia> c18 = em.createQuery("SELECT new SalaConferencia(c.tema, s.capacidad) FROM Sala s "
                + "JOIN s.conferencias c", SalaConferencia.class)
                .getResultList();
        System.out.println("(18) Nombre de la conferencia junto a la capacidad de la sala en la que se da (record): ");
        for (SalaConferencia scr : c18)
            System.out.println(" - { Conferencia: " + scr.nombConf () + ", capacidad: " + scr.capacidad() + " }");
    }
}
}
```

En lugar de escribir consultas como cadenas de texto (*JPQL*), el *API* `Criteria` permite crear consultas de forma dinámica mediante el uso de una *API* basada en clases y métodos.

4.5.2 Consultas programáticas.

Las consultas programáticas en *JPA* **se implementan a través de la *API* `Criteria`**, una herramienta poderosa para construir consultas de forma programática y tipada (basada en la programación funcional). La **interfaz `CriteriaBuilder`** desempeña un papel fundamental en este proceso.

Características principales de **`CriteriaBuilder`**:

- ✓ Forma parte de la especificación de *JPA*.
- ✓ **Ofrece una manera orientada a objetos para construir consultas, evitando la dependencia de lenguajes como *JPQL* o *SQL***.
- ✓ Permite **crear objetos `CriteriaQuery` que representan consultas** a la *BD*.

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<MiEntidad> criteriaQuery = criteriaBuilder.createQuery(MiEntidad.class);
```

Se deben importar las clases:

```
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
```

Resumen de **funcionalidades de `CriteriaBuilder`**:

- **Creación de consultas**: facilita la creación de consultas mediante objetos `CriteriaQuery`.
- **Tipado fuerte**: construye consultas tipadas, reduciendo errores en tiempo de ejecución al identificarlos en tiempo de compilación.
- **Consultas de selección**: permite recuperar entidades o campos específicos.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<MyEntity> query = cb.createQuery(MyEntity.class);
Root<MyEntity> root = query.from(MyEntity.class);
query.select(root);
```

- **Condiciones de filtrado**: crea cláusulas `WHERE` para filtrar resultados.

```
Predicate condition = cb.equal(root.get("nombre"), "valorBuscado");
query.where(condition);
```

- **Uniones (joins)**: permite realizar uniones entre entidades.

```
Join<MyEntity, OtherEntity> join = root.join("otraEntidad");
```

- **Funciones de agregación**: proporciona funciones como **`AVG`**, **`SUM`**, **`COUNT`**, etc.

```
Expression<Double> avgExpression = cb.avg(root.get("edad"));
```

- **Ordenamiento: permite especificar el orden de los resultados.**

```
query.orderBy(cb.asc(root.get("nombre")));
```

- **Parámetros y enlace de valores**: soporta el uso de parámetros y su enlace dinámico.

```
ParameterExpression<String> param = cb.parameter(String.class);
query.where(cb.equal(root.get("nombre"), param));
```

- **Subconsultas**: construye subconsultas para operaciones complejas.

```
Subquery<Integer> subquery = query.subquery(Integer.class);
Root<OtherEntity> subqueryRoot = subquery.from(OtherEntity.class);
subquery.select(cb.max(subqueryRoot.get("valor")));
query.where(cb.equal(root.get("otroCampo"), subquery));
```

Para obtener los resultados, se utiliza el método `createQuery` del `EntityManager`:

```
List<MiEntidad> results = entityManager.createQuery(criteriaQuery).getResultList();
```

Ejemplo: consultas sobre las tablas `sala` y `conferencia` de la *BD* `conferencias` haciendo uso de `CriteriaBuilder`.

- Dependencias (`pom.xml`) → Ver ejemplo punto 4.5.1.
- `Sala.java` → Ver ejemplo punto 4.5.1.
- `Conferencia.java` → Ver ejemplo punto 4.5.1.
- `persistence.xml` → Ver ejemplo punto 4.5.1.
- `ConsultasConferenciasCriteriaBuilder.java`

Componentes que ayudan a construir consultas de forma programática:
- `Root`: representa la entidad principal de la consulta (similar a una tabla).
- `Join`: relaciona entidades mediante sus asociaciones.
- `Predicate`: define condiciones lógicas para filtros (`WHERE`, `AND`, `OR`).
- `Subquery`: representa una subconsulta anidada dentro de la principal.
- `CriteriaBuilder`: punto de entrada para construir consultas.
- `CriteriaQuery`: estructura principal de la consulta (selección, filtros, orden).

```
import jakarta.persistence.*;
import jakarta.persistence.criteria.*;
import java.util.List;

public class ConsultasConferenciasCriteriaBuilder {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("conferencias");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();

// 1. Consultar conferencias por id
            CriteriaQuery<Conferencia> cq1 = cb.createQuery(Conferencia.class);
            Root<Conferencia> conferencia1 = cq1.from(Conferencia.class);
            cq1.where(cb.equal(conferencia1.get("codConferencia"), "POO1314"));
            Conferencia conf1 = entityManager.createQuery(cq1).getSingleResult();
            System.out.println("(1) Datos de la conferencia 'POO1314': ");
            System.out.println(conf1);

// 2. Consultar todas las conferencias
            CriteriaQuery<Conferencia> cq2 = cb.createQuery(Conferencia.class);
            Root<Conferencia> conferencia2 = cq2.from(Conferencia.class);
            cq2.select(conferencia2); // No necesaria esta línea, ya que se obtienen objetos completos de la clase
            List<Conferencia> conf2 = entityManager.createQuery(cq2).getResultList();
            System.out.println("(2) Todas las conferencias de la tabla: ");
            for (Conferencia conferencia : conf2)
                System.out.println(conferencia);

// 3. Consultar las conferencias que se dan en la sala 'Afrodita'
            CriteriaQuery<Conferencia> cq3 = cb.createQuery(Conferencia.class);
            Root<Conferencia> conferencia3 = cq3.from(Conferencia.class);
            Join<Conferencia, Sala> salaJoin3 = conferencia3.join("nombreSala");
            cq3.where(cb.equal(salaJoin3.get("nombreSala"), "Afrodita"));
            List<Conferencia> conf3 = entityManager.createQuery(cq3).getResultList();
```

```
            System.out.println("(3) Conferencias que se dan en la sala 'Afrodita': ");
            for (Conferencia conferencia : conf3)
                System.out.println(conferencia);
// 4. Obtener el precio medio de las conferencias del turno de mañana
            CriteriaQuery<Double> cq4 = cb.createQuery(Double.class);
            Root<Conferencia> conferencia4 = cq4.from(Conferencia.class);
            Predicate condicion4 = cb.equal(conferencia4.get("turno"), 'M');
            cq4.where(condicion4);
            cq4.select(cb.avg(conferencia4.get("precio")));
            double media = (double) entityManager.createQuery(cq4).getSingleResult();
            System.out.print("(4) Precio medio de las conferencias del turno de 'M': ");
            System.out.println(media);
// 5. Obtener el código y el tema de las conferencias cuyo precio esté comprendido entre 12 y 19, ordenado por
el tema de la conferencia de forma descendente
            CriteriaQuery<Object[]> cq5 = cb.createQuery(Object[].class);
            Root<Conferencia> conferencia5 = cq5.from(Conferencia.class);
            cq5.multiselect(conferencia5.get("codConferencia"), conferencia5.get("tema"));
            cq5.where(cb.between(conferencia5.get("precio"), 12, 19));
            cq5.orderBy(cb.desc(conferencia5.get("tema")));
            List<Object[]> conf5 = entityManager.createQuery(cq5).getResultList();
            System.out.println("(5) Cód. y tema de las conferencias entre 12 y 19, ordenadas por tema de forma
descendente: ");
            for (Object[] conferencia : conf5)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
// 6. Obtener el código y el tema de las conferencias cuyo precio esté comprendido entre 12 y 19, y se den en
la sala 'Afrodita', ordenado por el tema de la conferencia
            CriteriaQuery<Object[]> cq6 = cb.createQuery(Object[].class);
            Root<Conferencia> conferencia6 = cq6.from(Conferencia.class);
            Join<Conferencia, Sala> salaJoin6 = conferencia6.join("nombreSala");
            cq6.multiselect(conferencia6.get("codConferencia"), conferencia6.get("tema"));
            cq6.where(cb.between(conferencia6.get("precio"), 12, 19), cb.equal(salaJoin6.get("nombreSala"),
"Afrodita"));
            cq6.orderBy(cb.asc(conferencia6.get("tema")));
            List<Object[]> conf6 = entityManager.createQuery(cq6).getResultList();
            System.out.println("(6) Cód. y tema de las conferencias entre 12 y 19 que se dan en la sala
Afrodita ordenadas por tema de forma ascendente: ");
            for (Object[] conferencia : conf6)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
// 7. JOIN - Obtener el código, el tema de la conferencia y la capacidad de la sala para aquellas cuyo precio
esté comprendido entre 12 y 19
            CriteriaQuery<Object[]> cq7 = cb.createQuery(Object[].class);
            Root<Conferencia> conferencia7 = cq7.from(Conferencia.class);
            Join<Conferencia, Sala> salaJoin7 = conferencia7.join("nombreSala");
            cq7.multiselect(conferencia7.get("codConferencia"), conferencia7.get("tema"),
salaJoin7.get("capacidad"));
            cq7.where(cb.between(conferencia7.get("precio"), 12, 19));
            List<Object[]> conf7 = entityManager.createQuery(cq7).getResultList();
            System.out.println("(7) Cód., tema y capacidad de la sala de las conferencias cuyo precio está
comprendido entre 12 y 19: ");
            for (Object[] conferencia : conf7)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1] + " : " + conferencia[2]);
// 8. SUBCONSULTAS - Obtener el código y el tema de las conferencias que se dan en una sala con capacidad >= 200
            CriteriaQuery<Object[]> cq8 = cb.createQuery(Object[].class);
            Root<Conferencia> conferencia8 = cq8.from(Conferencia.class);
            Subquery<String> subconsulta = cq8.subquery(String.class);
            Root<Sala> subconsultaSala = subconsulta.from(Sala.class);
            subconsulta.select(subconsultaSala.get("nombreSala")).where(cb.greaterThanOrEqualTo(subconsultaSala
.get("capacidad"),200));
            cq8.multiselect(conferencia8.get("codConferencia"),
conferencia8.get("tema")).where(cb.in(conferencia8.get("nombreSala").get("nombreSala")).value(subconsulta));
            List<Object[]> conf8 = entityManager.createQuery(cq8).getResultList();
        System.out.println("(8) Cód. y tema de las conferencias que se dan en una sala con una capacidad >= 200: ");
            for (Object[] conferencia : conf8)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
        }
    }
}
```

El método **multiselect se utiliza para seleccionar múltiples columnas** o expresiones en una consulta. Esto permite crear una consulta que **devuelve datos en forma de `Object[]`** en lugar de instancias completas de entidades.

4.5.3 Consultas nativas SQL.

No se puede utilizar TypedQuery directamente con `createNativeQuery` en *JPA*.

El **método `createNativeQuery`** es parte de la *API* de persistencia de *JPA* y se utiliza para crear consultas *SQL* nativas (permite ejecutar consultas *SQL* directamente sobre la *BD*). **`createNativeQuery` devuelve un objeto de tipo `Query`**.
Ejemplo: consultas sobre las tablas `sala` y `conferencia` de la *BD* `conferencias` haciendo uso de consultas nativas.

- Dependencias (`pom.xml`) → Ver ejemplo punto 5.4.1.
- `Sala.java` → Ver ejemplo punto 5.4.1.
- `Conferencia.java` → Ver ejemplo punto 5.4.1.
- `src/main/resources/META-INF/persistence.xml` → Ver ejemplo punto 5.4.1.
- `ConsultasConferenciasNativas.java`

Se pueden pasar parámetros a una consulta *SQL* tanto con la sintaxis de signo de interrogación (?) como con la sintaxis de dos puntos (:). La sintaxis con : se utiliza para parámetros con nombre, mientras que la sintaxis con ? se usa para parámetros posicionales, donde se asignan valores según su posición en la consulta.

```
import jakarta.persistence.*;
import java.math.BigDecimal;
import java.util.List;

public class ConsultasConferenciasNativas {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("conferencias");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
// 1. Consultar conferencias por id
            Query queryCons1 = entityManager.createNativeQuery("SELECT * FROM conferencia WHERE cod_conferencia =
:codigo");
            queryCons1.setParameter("codigo", "POO1314");
            Conferencia conf1 = (Conferencia) queryCons1.getSingleResult();
            System.out.println("(1) Datos de la conferencia 'POO1314': ");
            System.out.println(conf1);
```

IMPORTANTE: el método `createNativeQuery` devuelve una instancia de `Query`, y no está diseñado para funcionar con `TypedQuery`.

```
// 2. Consultar todas las conferencias
            Query queryCons2 = entityManager.createNativeQuery("SELECT * FROM conferencia");
            List<Conferencia> conf2 = queryCons2.getResultList();
            System.out.println("(2) Todas las conferencias de la tabla: ");
            for (Conferencia conferencia : conf2)
                System.out.println(conferencia);
// 3. Consultar las conferencias que se dan en la sala 'Afrodita'
            Query queryCons3 = entityManager.createNativeQuery("SELECT * FROM conferencia WHERE nombre_sala =
:nombreSala");
            queryCons3.setParameter("nombreSala", "Afrodita");
            List<Conferencia> conf3 = queryCons3.getResultList();
            System.out.println("(3) Conferencias que se dan en la sala 'Afrodita': ");
            for (Conferencia conferencia : conf3)
                System.out.println(conferencia);
// 4. Obtener el precio medio de las conferencias del turno de mañana
            Query queryCons4 = entityManager.createNativeQuery("SELECT AVG(precio) AS media FROM conferencia WHERE
turno = :turno");
            queryCons4.setParameter("turno", 'M');
            BigDecimal media = (BigDecimal) queryCons4.getSingleResult();
            System.out.print("(4) Precio medio de las conferencias del turno de 'M': ");
            System.out.println(media);
// 5. Obtener el código y el tema de las conferencias cuyo precio esté comprendido entre 12 y 19, ordenado por el tema
de la conferencia
            Query queryCons5 = entityManager.createNativeQuery("SELECT cod_conferencia, tema FROM conferencia WHERE
precio BETWEEN :min AND :max ORDER BY tema");
            queryCons5.setParameter("min", 12);
            queryCons5.setParameter("max", 19);
            List<Object[]> conf5 = queryCons5.getResultList();
    System.out.println("(5) Cód. y tema de las conferencias entre 12 y 19, ordenadas por tema de forma ascendente: ");
            for (Object[] conferencia : conf5)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
// 6. Obtener el código y el tema de las conferencias cuyo precio esté comprendido entre 12 y 19, y se den en la sala
'Afrodita', ordenado por el tema de la conferencia
            Query queryCons6 = entityManager.createNativeQuery("SELECT cod_conferencia, tema FROM conferencia WHERE
precio BETWEEN :min AND :max AND nombre_sala = :sala ORDER BY tema");
            queryCons6.setParameter("min", 12);
            queryCons6.setParameter("max", 19);
            queryCons6.setParameter("sala", "Afrodita");
            List<Object[]> conf6 = queryCons6.getResultList();
            System.out.println("(6) Cód. y tema de las conferencias entre 12 y 19, y que se dan en la sala 'Afrodita'
ordenadas por tema de forma ascendente: ");
            for (Object[] conferencia : conf6)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
// 7. JOIN - Obtener el código, el tema de la conferencia y la capacidad de la sala para aquellas cuyo precio esté
comprendido entre 12 y 19
            Query queryCons7 = entityManager.createNativeQuery("SELECT cod_conferencia, tema, capacidad FROM
conferencia JOIN sala USING (nombre_sala) WHERE precio BETWEEN :min AND :max");
            queryCons7.setParameter("min", 12);
            queryCons7.setParameter("max", 19);
            List<Object[]> conf7 = queryCons7.getResultList();
            System.out.println("(7) Cód., tema y capacidad de la sala de las conferencias cuyo precio está comprendido
entre 12 y 19: ");
            for (Object[] conferencia : conf7)
```

```
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1] + " : " + conferencia[2]);
// 8. SUBCONSULTAS - Obtener el código y el tema de las conferencias que se dan en una sala con capacidad >= 200
            Query queryCons8 = entityManager.createNativeQuery("SELECT cod_conferencia, tema FROM conferencia WHERE
nombre_sala IN (SELECT nombre_sala FROM sala WHERE capacidad >= :capacidad)");
            queryCons8.setParameter("capacidad", 200);
            List<Object[]> conf8 = queryCons8.getResultList();
          System.out.println("(8) Cód. y tema de las conferencias que se dan en una sala con una capacidad >= 200: ");
            for (Object[] conferencia : conf8)
                System.out.println(" - " + conferencia[0] + " : " + conferencia[1]);
        }
    }
}
```

4.5.4 Consultas con nombre.

Las consultas con nombre permiten asignar un identificador único a una consulta *JPQL*, facilitando su reutilización, sin necesidad de escribir el mismo código repetidamente. Estas consultas **se definen directamente en las entidades, empleando anotaciones específicas**.

Características de las consultas con nombre:

- ✓ **Inmutabilidad**: una vez definidas, no se pueden modificar. Esto asegura su estabilidad y consistencia.
- ✓ **Conversión anticipada**: son leídas y traducidas a *SQL* durante la inicialización del contexto de persistencia, lo que mejora su eficiencia y rendimiento.
- ✓ **Definición flexible**: aunque suelen declararse con anotaciones en las clases entidad, también pueden configurarse mediante *XML*.

La **anotación `@NamedQuery` permite definir una consulta con nombre a través de los parámetros `name`** (nombre único de la consulta) **y `query`** (código *JPQL*):

```
@NamedQuery(
    name = "MiEntidad.encontrarPorNombre",
    query = "SELECT e FROM MiEntidad e WHERE e.nombre = :nombre"
)
```

Para **definir múltiples consultas con nombre en una misma entidad**, se utiliza la anotación **`@NamedQueries`**:

```
@NamedQueries({
    @NamedQuery(name = "MiEntidad.encontrarPorNombre", query = "SELECT e FROM MiEntidad e WHERE e.nombre = :nombre"),
    @NamedQuery(name = "MiEntidad.encontrarActivos", query = "SELECT e FROM MiEntidad e WHERE e.activo = true")
})
```

Las **consultas con nombre admiten parámetros, tanto posicionales como nominales**, para hacerlas más dinámicas.

El **nombre de la consulta debe ser único dentro de la unidad de persistencia**. Se recomienda el formato: `NombreEntidad.NombreConsulta`.

También es posible definir **consultas *SQL* nativas** mediante la anotación **`@NamedNativeQuery`**:

```
@NamedNativeQuery(
    name = "MiEntidad.nativeEncontrarPorNombre",
    query = "SELECT * FROM mi_tabla WHERE nombre = :nombre",
    resultClass = MiEntidad.class
)
```

Para resultados de varias tablas (uniones), utilizar un *DTO*. Esto da más flexibilidad para manejar los resultados sin requerir que coincidan exactamente con una sola entidad.

Estas consultas son **útiles para aprovechar funciones específicas del *SGBD* o para realizar consultas complejas no soportadas por *JPQL***. Para **definir varias en una misma entidad**, utilizar la anotación **`@NamedNativeQueries`**.

Ejemplo: consultas con nombre sobre la tabla `sala` de la *BD* `conferencias`.

- ➢ Dependencias (`pom.xml`) → Ver ejemplos anteriores.
- ➢ `src/main/resources/META-INF/persistence.xml` → Ver ejemplo punto 5.4.1.
- ➢ `Sala.java`

```
import jakarta.persistence.*;

@Entity
@Table(name = "sala")
@NamedQueries({
      @NamedQuery(name = "Sala.Todas", query = "SELECT s FROM Sala s"),
      @NamedQuery(name = "Sala.PorNombre", query = "SELECT s FROM Sala s WHERE s.nombreSala LIKE :nombreSala"),
      @NamedQuery(name = "Sala.CapacidadMayor175", query = "SELECT s FROM Sala s WHERE s.capacidad > :capacidad")
})
public class Sala {
    @Id
    @Column(name = "nombre_sala")
```

```
    private String nombreSala;
    @Column(name = "capacidad")
    private int capacidad;
    @Override
    public String toString() {
        return "Sala { " + "nombreSala = '" + nombreSala + "' , capacidad = " + capacidad + " }";
    }
}
```

➢ ConsultasConNombreSalas.java

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import java.util.List;
public class ConsultasConNombreSalas {
    public static void main(String[] args) {
        try (EntityManagerFactory emf = Persistence.createEntityManagerFactory("conferencias");
             EntityManager em = emf.createEntityManager()) {
            // Todas las salas
            List<Sala> salasTodas = em.createNamedQuery("Sala.Todas", Sala.class).getResultList();
            System.out.println("Todas las salas: ");
            for (Sala sala: salasTodas)
                System.out.println(sala);
            // Sala por nombre
            Sala salasNombre = em.createNamedQuery("Sala.PorNombre", Sala.class).setParameter("nombreSala",
                    "Afrodita").getSingleResult();
            System.out.println("Sala Afrodita: ");
            System.out.println(salasNombre);
            // Salas con capacidad superior a 175
            List<Sala> salasCapacidad = em.createNamedQuery("Sala.CapacidadMayor175",
                    Sala.class).setParameter("capacidad", 175).getResultList();
            System.out.println("Salas con capacidad mayor a 175: ");
            for (Sala sala: salasCapacidad)
                System.out.println(sala);
        }
    }
}
```

Ejemplo: consultas nativas con nombre sobre la tabla `sala` de la *BD* `conferencias`. Mantener todo igual que en el ejemplo anterior, salvo la parte de definición de las consultas en `Sala.java`:

```
@NamedNativeQueries({
    @NamedNativeQuery(name = "Sala.Todas", query = "SELECT * FROM sala", resultClass = Sala.class),
    @NamedNativeQuery(name = "Sala.PorNombre", query = "SELECT * FROM sala WHERE nombre_sala LIKE :nombreSala",
            resultClass = Sala.class),
    @NamedNativeQuery(name = "Sala.CapacidadMayor175", query = "SELECT * FROM sala WHERE capacidad >
            :capacidad", resultClass = Sala.class)
})
```

TAREA

7. **Crear una aplicación que, usando la entidad `Coche` del ejercicio 4 y los datos disponibles en la *BD*, solicite la matrícula del coche por teclado y muestre sus datos por pantalla (hacer uso de *JPQL* para recuperar los datos).**
8. **Crear un programa que a partir de las tablas mapeadas del ejercicio 2 (cliente, pedido y comercial), obtenga un listado de pedidos de un cliente (hacer uso de *JPQL* para recuperar los datos). El identificador del cliente se obtendrá como argumento de la línea de comandos.**
9. **Repetir los ejercicios 7 y 8 haciendo uso de consultas nativas *SQL*.**
10. **Repetir los ejercicios 7 y 8 haciendo uso de consultas con nombre.**

4.6 Eliminar un objeto persistente.

Una de las operaciones fundamentales que permite realizar *JPA* es la eliminación de objetos persistentes. **Cuando se elimina un objeto, se desencadena una operación de borrado en la *BD*** correspondiente a ese objeto.

JPA proporciona varios **métodos para eliminar objetos**, pero los más comunes son:

- ✓ **`remove(entity)`**: este método elimina un objeto que está siendo gestionado por el `EntityManager`.

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

```
em.getTransaction().begin();
em.remove(usuario);           // Suponiendo que 'usuario' es un objeto a eliminar
em.getTransaction().commit();
em.close();
```

- ✓ **Query de eliminación**: se puede utilizar una consulta *JPQL* para eliminar objetos que cumplan ciertos criterios.

```
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();
Query query = em.createQuery("DELETE FROM Usuario u WHERE u.edad > 30");
query.executeUpdate();
em.getTransaction().commit();
em.close();
```

El objeto a eliminar debe estar en estado managed (gestionado) **para poder ser eliminado directamente con `remove()`**. Si el objeto está en estado detached (desconectado), es necesario volver a adjuntarlo al `EntityManager` antes de eliminarlo. Esto se puede lograr mediante el uso de `find()`, `merge()` o consultas *JPQL*.

Si una entidad tiene relaciones con otras entidades, es importante considerar las cascadas de eliminación.

Ejemplo:

- ➢ Dependencias (`pom.xml`) → Ver punto 4.3.1.
- ➢ `src/main/resources/META-INF/persistence.xml` → Ver punto 4.3.1.
- ➢ `Pokemon.java` → Ver código en punto 3.2.
- ➢ `EliminarPokemon.java`

`contains()` comprueba si una entidad está gestionada por el `EntityManager`. Usarlo para evitar operaciones innecesarias (como `merge()` o `persist()`) y para confirmar si una entidad está vinculada antes de modificarla o eliminarla.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class EliminarPokemon {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("pokemon");
             EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            Pokemon pok = entityManager.find(Pokemon.class, 1); // ID del Pokémon que se desea eliminar es 1

            if (pok != null) {
                EntityTransaction tx = entityManager.getTransaction();
                tx.begin();
                try {
                    entityManager.remove(pok);
                    tx.commit();
                } catch (Exception e) {
                    if (tx != null && tx.isActive()) // Manejar excepción que pueda ocurrir durante la transacción
                        tx.rollback();
                    System.out.println("Error: " + e.getMessage());
                }
            } else
                System.out.println("El Pokémon no existe.");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```
if (entityManager.contains(pok))
    entityManager.remove(pok);
```

La **gestión del borrado en cascada en *JPA* permite especificar cómo las operaciones de borrado deben propagarse desde una entidad principal a las entidades asociadas**. Se puede **configurar el borrado en cascada utilizando las anotaciones `@OneToOne`/`@OneToMany`/`@ManyToOne`/`@ManyToMany` y la propiedad `cascade`**.

La combinación de `CascadeType.ALL` y `orphanRemoval = true` en una relación *JPA* indica un comportamiento de cascada completo que va más allá de las operaciones básicas de persistencia y actualización:

- ✓ **`CascadeType.ALL`**: esta opción **indica que todas las operaciones de cascada deben aplicarse a la entidad asociada**.
- ✓ **`orphanRemoval = true`**: esta opción **se aplica específicamente a las relaciones de tipo `OneToOne` o `OneToMany`**. Cuando `orphanRemoval` **está configurado como `true` y se elimina una entidad padre que tiene una entidad hija asociada, la entidad hija también se elimina automáticamente de la *BD***, incluso si no se ha eliminado directamente mediante `EntityManager.remove()`.

Ejemplo:

```
@Entity
public class Persona {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nombre;
    @OneToOne(cascade = CascadeType.ALL, mappedBy = "persona", orphanRemoval = true)
    private Direccion direccion;
    // getters y setters
}

@Entity
public class Direccion {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String calle;
    private String ciudad;
    @OneToOne
    @JoinColumn(name = "id_persona")
    private Persona persona;
    // getters y setters
}
```

En este ejemplo, si se elimina una instancia de `Persona`, también se eliminará automáticamente la instancia asociada de `Direccion` debido a la configuración de cascada (`CascadeType.ALL`). Además, si se asigna `null` a la propiedad `direccion` de una instancia de `Persona` y luego se realiza la operación de actualización (`EntityManager.merge()`), la instancia de `Direccion` se eliminará de la *BD* debido a `orphanRemoval = true`.

4.7 Modificar un objeto.

La modificación de objetos con *JPA* es una operación fundamental en el desarrollo de aplicaciones *Java*.

Cuando se modifica un objeto que está siendo gestionado por el `EntityManager` (en estado `managed`), **los cambios realizados en el objeto se reflejan automáticamente en la *BD* correspondiente cuando se realiza un `commit` a la transacción**.

El **estado de un objeto en *JPA* determina cómo se gestionan las actualizaciones**:

- ✓ **Objetos gestionados** (managed): están **asociados a un `EntityManager`**. Los **cambios realizados en ellos se rastrean automáticamente y se persisten al hacer `commit()` o al ejecutar un `flush()`. Se obtienen cuando se persiste una entidad con `persist()` o se recupera un objeto desde la *BD* con métodos como `find()` o `createQuery()`**.
- ✓ **Objetos desapegados** (detached): **no están asociados a ningún `EntityManager`**. Para **actualizarlos y reincorporarlos al contexto de persistencia, se utiliza el método `merge()`**.
- ✓ **Consultas de actualización**: permiten **modificar directamente la *BD* mediante *JPQL***. Son útiles para actualizaciones masivas o cuando no es necesario cargar los objetos en memoria.

`merge()` es un método de *JPA* que permite actualizar o crear entidades en la *BD*. Si la entidad que se pasa como argumento **ya existe en la *BD*, `merge()` actualiza sus propiedades**. Si la entidad **no existe, `merge()` crea un nuevo registro**. En ambos casos, `merge()` **devuelve una referencia al objeto gestionado por el `EntityManager`**, lo que permite realizar operaciones adicionales sobre el objeto.

La actualización en cascada se configura utilizando la anotación `@CascadeType` en las relaciones entre entidades. Por ejemplo, si una entidad `Pedido` tiene una relación de uno a muchos con la entidad `DetallePedido`, se puede configurar la actualización en cascada de la siguiente manera:

```
@Entity
public class Pedido {
    @OneToMany(cascade = CascadeType.ALL)
    private List<DetallePedido> detalles = new ArrayList<>();
    // ...
}
```

JPA define varios **tipos de operaciones en cascada**:

- ✓ **`PERSIST`**: persiste las entidades relacionadas cuando se persiste la entidad principal.
- ✓ **`MERGE`**: actualiza las entidades relacionadas cuando se actualiza la entidad principal.
- ✓ **`REMOVE`**: elimina las entidades relacionadas cuando se elimina la entidad principal.

- ✓ **REFRESH**: refresca las entidades relacionadas con los datos de la base de datos.
- ✓ **DETACH**: desasocia las entidades relacionadas del contexto de persistencia.
- ✓ **ALL**: incluye todas las operaciones anteriores.

Ejemplo:

- ➢ Dependencias (`pom.xml`) → Ver punto 4.3.1.
- ➢ `src/main/resources/META-INF/persistence.xml` → Ver punto 4.3.1.
- ➢ `Pokemon.java` → Ver código en punto 3.2. Copiar el mismo código y **añadir los métodos setter**.
- ➢ `ModificarPokemon.java`

```
import jakarta.persistence.*;
public class ModificarPokemon {
    public static void main(String[] args) {
        try (EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("pokemon");
            EntityManager entityManager = entityManagerFactory.createEntityManager()) {
            Pokemon pok = entityManager.find(Pokemon.class, 3); // ID del Pokémon que se desea modificar es 3
            if (pok != null) {
                EntityTransaction tx = entityManager.getTransaction();
                tx.begin();
                try {
                    pok.setNombre("Nuevo nombre");
                    pok.setDescripcion("Descripción modificada!!!");
                    // No se necesita hacer nada específico aquí, simplemente al haber obtenido el objeto de la BD
                    // y modificarlo dentro de una transacción, los cambios se reflejarán en la BD al hacer commit.
                    tx.commit();
                } catch (Exception e) {
                    if (tx != null && tx.isActive()) // Manejar excepción que pueda ocurrir en la transacción
                        tx.rollback();
                    System.out.println("Error: " + e.getMessage());
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```
// Crear y guardar un objeto gestionado
em.getTransaction().begin();
Persona persona = new Persona();
persona.setNombre("Juan");
em.persist(persona);
em.getTransaction().commit();
// Cerrar el EntityManager (objeto pasa a estado Detached)
em.close();

persona.setNombre("Juan Pérez"); // Modificar el objeto desapegado
em = emf.createEntityManager(); // Abrir un nuevo EntityManager
em.getTransaction().begin();
// No se usa merge aquí, por lo que el cambio no se persiste
em.getTransaction().commit();
```

4.8 Vaciar el contexto de persistencia.

El **método `clear()`** de la interfaz `EntityManager` en *JPA* **se utiliza para vaciar el contexto de persistencia, eliminando todas las entidades que se encuentran gestionadas en dicho contexto**. Esto significa que **las entidades que estaban siendo rastreadas por el `EntityManager` se desasocian, y cualquier cambio que no se haya sincronizado previamente con la *BD* será descartado**.

Características principales de `clear()`:

- ✓ **Desasocia entidades gestionadas**: todas las entidades que estaban en estado `managed` pasan a estado `detached`. Los cambios realizados en estas entidades después de la última sincronización se pierden.
- ✓ **No afecta a la *BD***: no realiza ninguna operación en la *BD*; simplemente afecta al contexto de persistencia en memoria.
- ✓ **Reinicia el contexto de persistencia**: es útil en escenarios donde se desea liberar memoria o evitar conflictos al trabajar con un gran número de entidades.

Nota: si hay cambios pendientes que no han sido sincronizados (`flush()`), estos se perderán tras llamar a `clear()`.

Ejemplo:

```
import jakarta.persistence.*;
@Entity
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    public Estudiante() {}
    public Estudiante(String nombre) { this.nombre = nombre; }
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    @Override
    public String toString() { return "Estudiante{id=" + id + ", nombre='" + nombre + "'}"; }
```

La diferencia entre `detach` y `clear` radica en su alcance. **`detach` desvincula una entidad específica del contexto de persistencia**, lo que significa que los cambios en esa entidad ya no se reflejarán en la *BD*, mientras que **`clear` elimina todas las entidades gestionadas por el `EntityManager`**, liberando el seguimiento de todas las entidades sin necesidad de especificar cuál.

```
            }
    import jakarta.persistence.*;
    public class ClearEjemplo {
        public static void main(String[] args) {
            EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUnidadPersistencia");
            EntityManager em = emf.createEntityManager();
            try {
                // 1. Insertar un estudiante en la BD
                em.getTransaction().begin();
                Estudiante estudiante = new Estudiante("Juan Pérez");
                em.persist(estudiante);
                em.getTransaction().commit();
                System.out.println("Estudiante persistido: " + estudiante);
                // 2. Modificar la entidad
                em.getTransaction().begin();
                Estudiante encontrado = em.find(Estudiante.class, estudiante.getId());
                System.out.println("Estudiante encontrado: " + encontrado);
                encontrado.setNombre("Carlos López"); // Cambiar el nombre
                em.clear(); // Vaciar el contexto de persistencia, entidad 'encontrado' pasa a estado detached
                // Intentar hacer cambios después de clear() no tiene efecto
                encontrado.setNombre("Nombre perdido");
                em.getTransaction().commit();
                System.out.println("Cambios no persistidos después de clear(): " + encontrado);
                // 3. Comprobar que no se guardaron los cambios
                em.getTransaction().begin();
                Estudiante comprobado = em.find(Estudiante.class, estudiante.getId());
                System.out.println("Estado en la BD: " + comprobado);
                em.getTransaction().commit();
            } catch (Exception e) {
                System.out.println("Error: " + e.getMessage());
                if (em.getTransaction().isActive())
                    em.getTransaction().rollback();
            } finally {
                em.close(); emf.close();
            }
        }
    }
```

4.9 Sentencias INSERT, UPDATE y DELETE.

4.9.1 Con JPQL.

En *JPQL*, las **operaciones de inserción, actualización y eliminación no se suelen realizar directamente** como en *SQL*. En lugar de eso, **se utiliza la persistencia de objetos para agregar nuevos registros, modificar los existentes o bien eliminarlos de la *BD***.

Para **operaciones simples y regulares en objetos individuales, la persistencia de objetos es generalmente más limpia y orientada a objetos**. Para **operaciones masivas o complejas a nivel de *BD*, las sentencias *JPQL* pueden ser más eficientes**.

Se utilizará el método `executeUpdate()` que **devuelve el número de filas a las que la consulta ha afectado**.

```
EntityManager em = Persistence.createEntityManagerFactory("nombre_up").createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
// INSERT --> NO SOPORTADO POR JPQL --> SE PODRÍAN UTILIZAR CONSULTAS NATIVAS SQL
// UPDATE
Query queryUpdate = em.createQuery("UPDATE Entidad e SET e.campo1 = :nuevoVal WHERE e.campo2 = :cond");
queryUpdate.setParameter("nuevoVal", nuevoValor);
queryUpdate.setParameter("cond", condicion);
queryUpdate.executeUpdate();
// DELETE
Query queryDelete = em.createQuery("DELETE FROM Entidad e WHERE e.campo1 = :condicion");
queryDelete.setParameter("condicion", condicion);
queryDelete.executeUpdate();
tx.commit();
em.close();
```

JPQL tiene una sintaxis concreta para `SELECT`, `UPDATE` y `DELETE`, pero **no tiene soporte directo para `INSERT INTO`** como en *SQL* tradicional. Se realiza mediante `entityManager.persist()` o haciendo uso de consultas nativas.

4.9.2 Con SQL nativo.

Si se desean realizar operaciones utilizando *SQL* nativo con *JPA*, **se puede hacer a través de la clase `Query` (`createNativeQuery`)**.

```
EntityManager entityManager = Persistence.createEntityManagerFactory("nombre_up").createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

// INSERT
String sqlInsert = "INSERT INTO tabla (columna1, columna2) VALUES (?, ?)";
Query queryInsert = entityManager.createNativeQuery(sqlInsert);
queryInsert.setParameter(1, valor1);
queryInsert.setParameter(2, valor2);
queryInsert.executeUpdate();

// UPDATE
String sqlUpdate = "UPDATE tabla SET columna1 = ? WHERE columna2 = ?";
Query queryUpdate = entityManager.createNativeQuery(sqlUpdate);
queryUpdate.setParameter(1, nuevoValor);
queryUpdate.setParameter(2, condicion);
queryUpdate.executeUpdate();

// DELETE
String sqlDelete = "DELETE FROM tabla WHERE columna1 = ?";
Query queryDelete = entityManager.createNativeQuery(sqlDelete);
queryDelete.setParameter(1, condicion);
queryDelete.executeUpdate();

transaction.commit();
entityManager.close();
```

Recordar que se pueden pasar parámetros tanto con la sintaxis de signo de interrogación (?) como con la sintaxis de dos puntos (:).

TAREA

11. Crear una aplicación que haga uso de *JPA* para gestionar una flota de taxis. La aplicación tiene que almacenar los siguientes datos:

- **De cada taxista: nombre, *DNI* y fecha de nacimiento (+ campos que se puedan necesitar).**
- **De cada taxi: precio, matrícula y número de plazas (+ campos que se puedan necesitar).**

Cuando empieza el turno de un taxista se le asigna uno de los taxis que no está siendo utilizado por nadie. Cada taxista, durante su jornada laboral, y hasta que esta concluya, será responsable del taxi asignado. Cuando finaliza el trabajo de un taxista, devuelve el taxi utilizado, que estará libre para asignarlo a otro trabajador.

La aplicación tendrá el siguiente menú:

1. **Alta de nuevo taxista.**
2. **Alta de nuevo taxi.**
3. **Comienzo de la jornada taxista.**
4. **Fin de la jornada taxista.**
5. **Información de un taxista y su taxi.**
6. **Mostrar taxistas trabajando.**
7. **Mostrar taxistas fuera de servicio.**
8. **Salir.**

Para facilitar el trabajo de los usuarios de la aplicación, cada vez que se solicite un *DNI* o una matrícula, se mostrará un listado con los datos disponibles. Por ejemplo, cuando se quiere finalizar una jornada de un taxista, antes de pedir el *DNI*, se puede mostrar todos los taxistas que están trabajando.

5 Eventos del ciclo de vida en *JPA*.

Los eventos del ciclo de vida de las entidades en *JPA* **permiten ejecutar lógica personalizada en puntos específicos del ciclo de vida de una entidad** (creación, actualización, eliminación, etc.). Esto es **útil para realizar validaciones, auditorías, inicialización de datos o cualquier lógica adicional relacionada con los datos**.

JPA proporciona seis **eventos principales** del ciclo de vida:

Evento	Anotación	Momento de ejecución
Antes de persistir	`@PrePersist`	Antes de que una entidad se almacene por primera vez en la *BD*.

Después de persistir	**`@PostPersist`**	Justo después de que una entidad se haya almacenado en la *BD*.
Antes de actualizar	**`@PreUpdate`**	Antes de que una entidad gestionada se actualice en la *BD*.
Después de actualizar	**`@PostUpdate`**	Justo después de que una entidad gestionada se haya actualizado en la *BD*.
Antes de eliminar	**`@PreRemove`**	Antes de que una entidad gestionada sea eliminada de la *BD*.
Después de eliminar	**`@PostRemove`**	Justo después de que una entidad gestionada se haya eliminado de la *BD*.
Cargada desde la *BD*	**`@PostLoad`**	Después de que una entidad se cargue desde la *BD*.

Cómo usar los eventos del ciclo de vida

- **Definir métodos para eventos**. Los métodos que respondan a los eventos deben ser: públicos o protegidos, sin retorno (`void`) y sin parámetros.
- **Anotar el método con el evento correspondiente**. Se usa la anotación adecuada para el evento deseado.

Consideraciones importantes:

- ✓ Los eventos como **`@PrePersist` y `@PreUpdate` se ejecutan dentro de la transacción, lo que significa que cualquier excepción lanzada anula la operación**.
- ✓ **Métodos anotados con eventos no deben realizar operaciones de larga duración**, ya que afectan el rendimiento.
- ✓ **`@PostLoad` no se ejecuta cuando se usan consultas nativas**, a menos que las entidades se pasen a un contexto gestionado.

Ejemplo: suponer una entidad `Usuario` con un campo `fechaCreacion` y `ultimaActualizacion`.

```
import jakarta.persistence.*;
import java.time.LocalDateTime;

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

    private String nombre;
    private LocalDateTime fechaCreacion;
    private LocalDateTime ultimaActualizacion;

    // Constructor, getters y setters

    @PrePersist
    public void antesDePersistir() {
        fechaCreacion = LocalDateTime.now();
        ultimaActualizacion = fechaCreacion;
        System.out.println("Se está persistiendo la entidad Usuario.");
    }

    @PreUpdate
    public void antesDeActualizar() {
        ultimaActualizacion = LocalDateTime.now();
        System.out.println("Se está actualizando la entidad Usuario.");
    }

    @PostLoad
    public void despuesDeCargar() {
        System.out.println("Entidad Usuario cargada desde la BD: " + this.nombre);
    }
}
```

6 Procedimientos y funciones almacenadas.

En *JPA*, existen **diferentes formas de invocar procedimientos almacenados y funciones almacenadas** definidas en la *BD*, lo que permite aprovechar la lógica implementada directamente en la *BD* para mejorar el rendimiento o simplificar tareas complejas. Puedes **invocar procedimientos almacenados** utilizar la clase **`StoredProcedureQuery`** de manera dinámica o bien emplear la anotación **`@NamedStoredProcedureQuery`** para definir consultas más estáticas. Para **funciones almacenadas**, se invoca la función como una consulta (**`Query`**) que retorna un valor.

6.1 Uso de `@NamedStoredProcedureQuery` (definido en la entidad).

La anotación **`@NamedStoredProcedureQuery` define procedimientos almacenados en la *BD* que pueden ser llamados desde *JPA***. Esta anotación **se aplica a nivel de clase de entidad y permite declarar**:

- ✓ El **nombre lógico del procedimiento** almacenado en la entidad (**name**).
- ✓ El **nombre del procedimiento** almacenado (**procedureName**).
- ✓ Los **parámetros que el procedimiento almacenado utiliza** (**parameters**).
- ✓ Los **parámetros de entrada y salida** (**@StoredProcedureParameter**).
  - El **modo de los parámetros** (**ParameterMode**):
    - Entrada (**IN**): se envían al procedimiento o función.
    - Salida (**OUT**): devuelven valores desde el procedimiento.
    - Entrada y salida (**INOUT**): sirven tanto para enviar como para recibir valores.
  - El **nombre de los parámetros** (**name**).
  - El **tipo de dato que espera el parámetro** (`Integer.class`, `Docuble.class`, etc.).

Ejemplo: suponer que se tiene un procedimiento almacenado en la *BD* llamado `calcular_salario` que calcula el salario total de un empleado dado su `ID`.

- Procedimiento almacenado.

```
DELIMITER //
CREATE PROCEDURE calcular_salario (IN empleado_id INT, OUT salario_total DECIMAL(10, 2))
BEGIN
    SELECT SUM(salario_base + bonificacion)
    INTO salario_total
    FROM Empleados
    WHERE id = empleado_id;
END //
DELIMITER ;
```

- Entidad con `@NamedStoredProcedureQuery`.

```
import jakarta.persistence.*;
@Entity
@NamedStoredProcedureQuery(
    name = "Empleado.calcularSalario",
    procedureName = "calcular_salario",
    parameters = {
        @StoredProcedureParameter(mode = ParameterMode.IN, name = "empleado_id", type = Integer.class),
        @StoredProcedureParameter(mode = ParameterMode.OUT, name = "salario_total", type = Double.class)
    }
)
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private Double salarioBase;
    private Double bonificacion;
    // Getters y setters
}
```

- Llamada al procedimiento almacenado.

```
EntityManager em = emf.createEntityManager();
StoredProcedureQuery query = em.createNamedStoredProcedureQuery("Empleado.calcularSalario");
query.setParameter("empleado_id", 1); // ID del empleado
query.execute();
System.out.println("El salario total es: " + query.getOutputParameterValue("salario_total"));
```

6.2 Uso de `StoredProcedureQuery` (en tiempo de ejecución).

El uso de `StoredProcedureQuery` en *JPA* está destinado a **invocar procedimientos almacenados** definidos en la *BD*. Proporciona una forma estandarizada de trabajar con procedimientos que aceptan parámetros de entrada y/o devuelven resultados (como parámetros de salida o conjuntos de resultados).

Pasos:

1. Crear objeto `StoredProcedureQuery`.
2. Registrar parámetros: uso de `registerStoredProcedureParameter`. Los parámetros pueden ser de entrada (`ParameterMode.IN`), salida (`ParameterMode.OUT`), o ambos (`ParameterMode.INOUT`).
3. Establecer valores de entrada: uso de `setParameter` para proporcionar valores a los parámetros de entrada.

4. Ejecutar el procedimiento: uso de `execute` para invocar el procedimiento almacenado.
5. Obtener valores de salida: uso de `getOutputParameterValue` para recuperar los resultados de los parámetros de salida.

Ejemplo: igual al anterior haciendo uso de `StoredProcedureQuery`.

- Llamada al procedimiento almacenado.

```
StoredProcedureQuery query = em.createStoredProcedureQuery("calcular_salario");
query.registerStoredProcedureParameter(1, Integer.class, ParameterMode.IN);
query.registerStoredProcedureParameter(2, Double.class, ParameterMode.OUT);
query.setParameter(1, 1);
query.execute();
System.out.println("El salario total es: " + query.getOutputParameterValue(2));
```

Se puede utilizar el nombre en lugar de la posición para registrar y establecer los valores de los parámetros.

6.3 Uso de `Query` para ejecutar funciones almacenadas.

Para **ejecutar funciones almacenadas** que devuelven un valor, el enfoque más sencillo es **usar consultas *SQL* nativas** en *JPA*.

Ejemplo: suponer que se tiene una función en la *BD* llamada `calcular_impuesto`, que calcula el impuesto para un producto dado su precio y su tasa de impuesto.

- Función almacenada.

```
DELIMITER //
CREATE FUNCTION calcular_impuesto (precio DECIMAL(10, 2), tasa DECIMAL(5, 2)) RETURNS DECIMAL(10, 2)
DETERMINISTIC
BEGIN
    RETURN precio * (tasa / 100);
END//
DELIMITER ;
```

- Llamar a la función usando consultas nativas.

```
import jakarta.persistence.*;
public class LlamarFuncion {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUnidadPersistencia");
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            Query query = em.createNativeQuery("SELECT calcular_impuesto(:precio, :tasa)");
            query.setParameter("precio", 100.0);
            query.setParameter("tasa", 21.0);
            System.out.println("El impuesto calculado es: " + query.getSingleResult());
            em.getTransaction().commit();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            em.close();
            emf.close();
        }
    }
}
```

7 Gestión de concurrencia y bloqueos.

Las **transacciones en *JPA* proporcionan una forma de manejar la consistencia y el aislamiento de las operaciones sobre la *BD*, pero no resuelven por sí solas los problemas de concurrencia o de bloqueos**. Las transacciones garantizan que las operaciones se ejecuten de manera atómica (todas las operaciones dentro de la transacción se completan correctamente o ninguna se aplica), pero los **problemas de concurrencia, siguen existiendo si múltiples transacciones intentan modificar los mismos datos simultáneamente**.

Para **solucionar estos problemas**, se usan **técnicas** como:

- ✓ **Bloqueo optimista**: las **transacciones no bloquean los datos de la *BD*, pero cada transacción controla si el valor de los datos que está modificando ha cambiado desde que fue leído, usando un campo de versión**

(con **@Version**). **Si el valor ha cambiado entre la lectura y la escritura, se genera una excepción** (**OptimisticLockException**), lo que indica que otro proceso ya ha modificado esos datos.

- ✓ **Bloqueo pesimista**: las **transacciones bloquean físicamente los registros mientras se modifican para que otras transacciones no puedan acceder a esos registros**. Esto se puede hacer con **LockModeType.PESSIMISTIC_WRITE** o **LockModeType.PESSIMISTIC_READ** para bloquear los datos hasta que la transacción termine.

El método **refresh() se utiliza para actualizar el estado de una entidad desde la *BD*, sobrescribiendo cualquier cambio no guardado que haya hecho el programa en la entidad gestionada por el contexto de persistencia** (el EntityManager). Esto es útil cuando es necesario asegurarse de que los datos de la entidad reflejan el estado actual de la *BD*, especialmente en escenarios de concurrencia o cuando otras transacciones pueden haber modificado los datos.

Ejemplo: bloqueo optimista.

- Producto.java

```java
import jakarta.persistence.*;
@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private int cantidad;
    @Version // Campo para manejar concurrencia optimista
    private int version;
    // Getters y setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public int getCantidad() { return cantidad; }
    public void setCantidad(int cantidad) { this.cantidad = cantidad; }
    public int getVersion() { return version; }
    public void setVersion(int version) { this.version = version; }
}
```

- OptimisticLockingEjemplo.java

```java
import jakarta.persistence.*;
public class OptimisticLockingEjemplo {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUnidadPersistencia");
        EntityManager em1 = emf.createEntityManager();
        EntityManager em2 = emf.createEntityManager();
        try {
            // Transacción 1: Obtener y actualizar un producto
            em1.getTransaction().begin();
            Producto producto1 = em1.find(Producto.class, 1L);
            producto1.setCantidad(producto1.getCantidad() - 10);
            producto1.setNombre("Producto actualizado por Empleado 1");
            // Simular que otra transacción hace cambios en el producto (Transacción 2)
            em2.getTransaction().begin();
            Producto producto2 = em2.find(Producto.class, 1L);
            producto2.setCantidad(producto2.getCantidad() - 5);
            producto2.setNombre("Producto actualizado por Empleado 2");
            // Guardar cambios de la primera transacción
            em1.getTransaction().commit();
            try { // Intentar guardar cambios de la segunda transacción
                em2.getTransaction().commit();
            } catch (OptimisticLockException e) {
                System.out.println("Conflicto detectado: " + e.getMessage());
                em2.getTransaction().rollback(); // Rollback en caso de conflicto
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            em1.close(); em2.close(); emf.close();
        }
    }
}
```

Ejemplo: bloqueo pesimista.

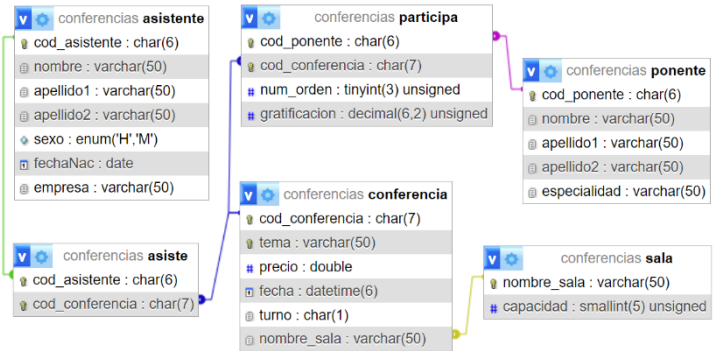
- Producto.java → Ver código de ejemplo anterior.

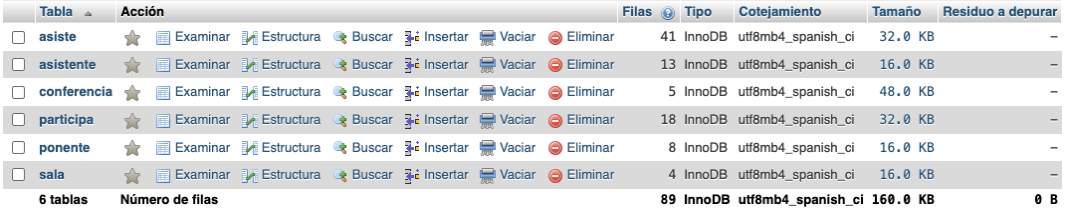
➢ `PessimisticLockingEjemplo.java`

```
import jakarta.persistence.*;

public class PessimisticLockingEjemplo {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUnidadPersistencia");
        EntityManager em1 = emf.createEntityManager();
        EntityManager em2 = emf.createEntityManager();

        try {
            // Transacción 1: Obtener y actualizar el producto con bloqueo pesimista
            em1.getTransaction().begin();
            Producto producto1 = em1.find(Producto.class, 1L, LockModeType.PESSIMISTIC_WRITE);

            // La transacción 1 tiene el bloqueo exclusivo en el registro.
            System.out.println("Transacción 1 obtuvo bloqueo en el producto con ID: " + producto1.getId());
            producto1.setCantidad(producto1.getCantidad() - 10); // Realizar la actualización

            new Thread(() -> { // Simular una segunda transacción que intenta acceder al mismo producto
                em2.getTransaction().begin();
                Producto producto2 = em2.find(Producto.class, 1L, LockModeType.PESSIMISTIC_WRITE);
                try {
                    Thread.sleep(2000); // Esperar un tiempo para simular un retraso en la transacción 2
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                producto2.setCantidad(producto2.getCantidad() - 5);
                em2.getTransaction().commit(); // Se liberará el bloqueo después de commit
                System.out.println("Transacción 2 completada con éxito.");
            }).start();

            Thread.sleep(500); // Esperar un poco para que la transacción 2 intente obtener el bloqueo

            // Ahora, cuando la transacción 2 intente obtener el bloqueo, no podrá
            // y quedará bloqueada hasta que transacción 1 haga commit.
            em1.getTransaction().commit(); // Al commit de em1 se libera el bloqueo
            System.out.println("Transacción 1 completada.");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            em1.close(); em2.close(); emf.close();
        }
    }
}
```

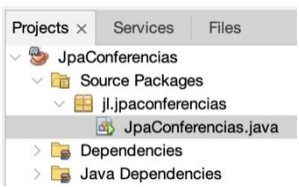
Anexo I: *JPA* y *Apache NetBeans*.

En este punto, se utilizará la *BD* "conferencias", que está disponible para su descarga en la plataforma Moodle Centros. Esta *BD* contiene las tablas y datos necesarios para trabajar con la gestión de conferencias. El acceso a la *BD* se realizará utilizando el usuario conferencias y la clave conferencias. Asegurarse de descargar e importar correctamente la BD en el *SGBD* antes de comenzar con los siguientes apartados.

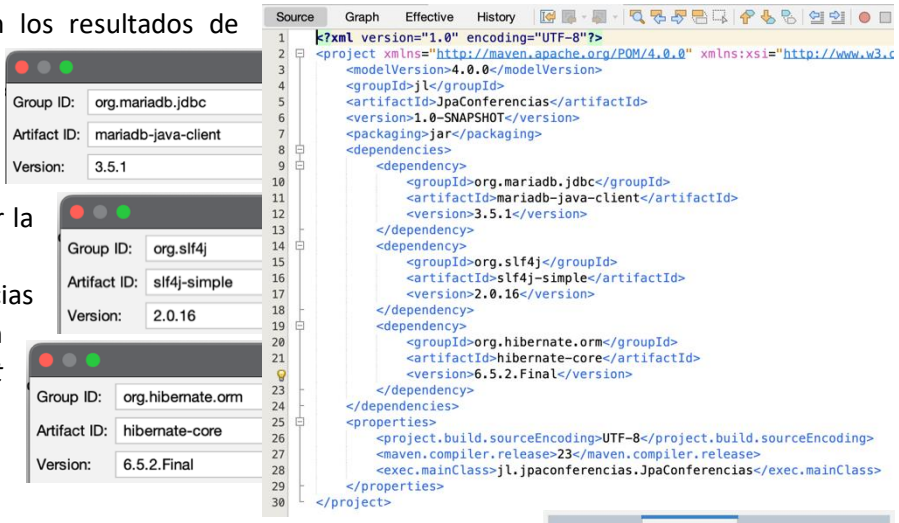




1. **Crear un nuevo proyecto *Java* con *Maven*. Nombrarlo como `JpaConferencias`.**
2. **Añadir dependencias al proyecto.**
   - Hacer clic derecho sobre la carpeta *Dependencies* en el árbol del proyecto (ubicada en la raíz del proyecto *Maven* en *NetBeans*).
   - Seleccionar la opción *Add Dependency…* en el menú contextual.

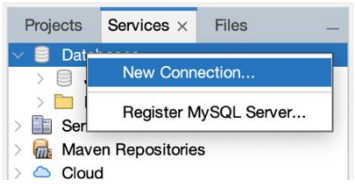


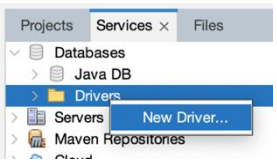
- En la ventana emergente, introducir el *Group ID*, *Artifact ID*, y opcionalmente la *Version* de la dependencia que se desea agregar. Si se no conocen exactamente estos datos, utilizar la pestaña *Search* y el campo de texto de búsqueda *Query* para buscar en el repositorio central de *Maven*.
- Una vez localizada la dependencia en los resultados de búsqueda, seleccionarla.
- Hacer clic en *Add* para agregarla al archivo `pom.xml` del proyecto. *NetBeans* actualizará automáticamente el archivo `pom.xml` para incluir la dependencia seleccionada.
- Para asegurarse de que las dependencias estén disponibles, hacer clic derecho en el proyecto y seleccionar *Reload Project* o *Clean and Build*.

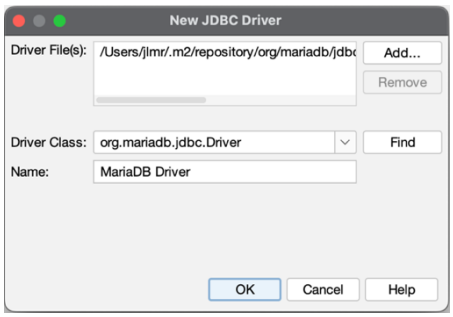


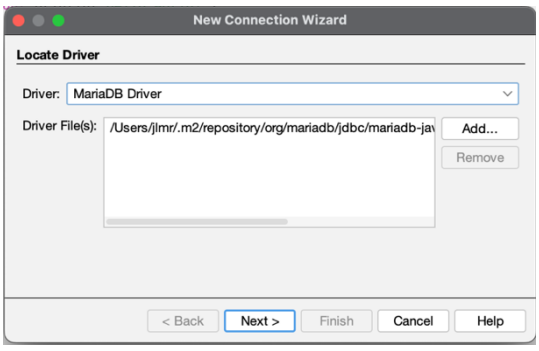
3. Crear conexión con la BD.

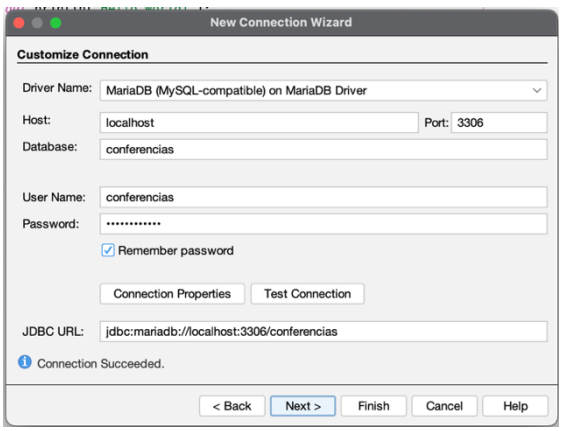
- En la barra de herramientas de *NetBeans*, seleccionar *Window* → *Services* si aún no está visible. Una forma más sencilla es utilizar la pestaña *Services*, ubicada en el panel lateral del entorno de desarrollo.
- En el panel *Services*, localizar la sección *Databases* y hacer clic derecho, seleccionar *New Connection*.
- Aparecerá el asistente para configurar la conexión.
- En el asistente, seleccionar el controlador correspondiente. Si el controlador no aparece en la lista:
  - En el panel *Services*, expandir la sección *Databases*.
  - Hacer clic derecho sobre *Drivers* y selecciona *New Driver*.
  - Seleccionar el archivo *JAR* del controlador. En la ventana que aparece, haz clic en *Add* y buscar/seleccionar el archivo *JAR* del controlador *JDBC* de *MariaDB*. Si ya se ha agregado la dependencia de *MariaDB* al proyecto *Maven*, el archivo *JAR* correspondiente se descarga automáticamente al repositorio local. En *Windows*, el repositorio local suele ubicarse en: `C:\Users\<tu_usuario>\.m2\repository`. Dentro del repositorio, las dependencias están organizadas por el *Group ID*. Para *MariaDB*, la ruta típica es: `C:\Users\<tu_usuario>\.m2\repository\org\mariadb\mariadb-java-client`.
  - Hacer clic en *Open* para agregarlo.
  - En el campo *Driver Class*, introducir la clase del controlador de *MariaDB*: `org.mariadb.jdbc.Driver`
  - Proporcionar un nombre descriptivo para el controlador, por ejemplo, *MariaDB Driver*.
  - Hacer clic en *OK* para guardar el nuevo controlador en la lista.
- Confirmar la selección del controlador y hacer clic en *Next >*.
- En la siguiente ventana completar los datos.





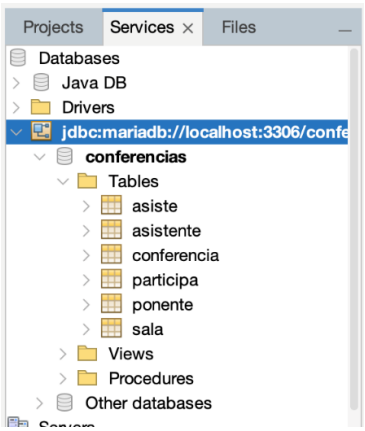






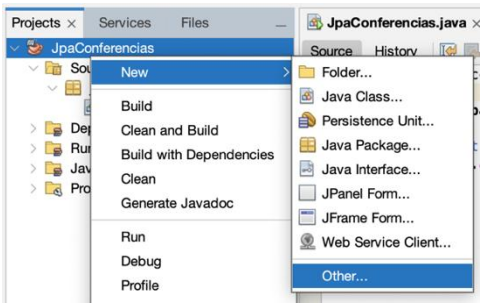
- Cambiar localhost y 3306 si se está utilizando otro host o puerto.
- User Name: introducir el usuario de la *BD*, en este caso, conferencias.
- Password: introducir la contraseña del usuario, en este caso, conferencias.

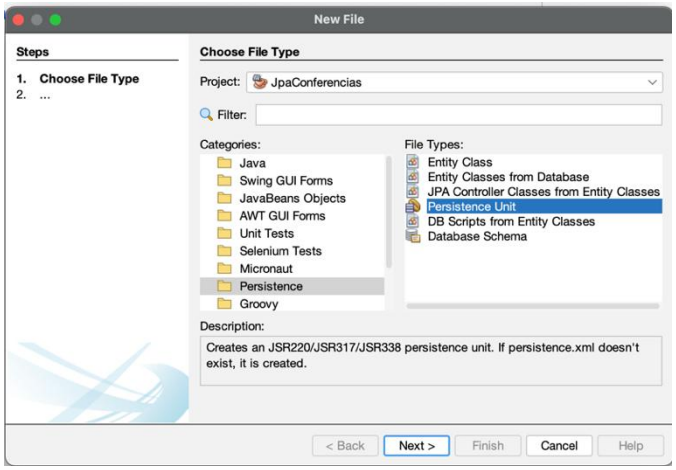
- Hacer clic en el botón *Test Connection* para comprobar que los detalles son correctos y que *NetBeans* puede conectarse a la *BD*.
- Si la prueba es exitosa, hacer clic en *Next >*.
- No cambiar nada y hacer nuevamente clic en *Next >*.
- No cambiar nada y hacer clic en *Finish*.
- Una vez creada la conexión, esta aparecerá en la sección *Databases* del panel *Services*.
- Hacer clic en el símbolo de > junto a la conexión para expandirla.
- Navegar por las tablas y otros objetos de la *BD*.



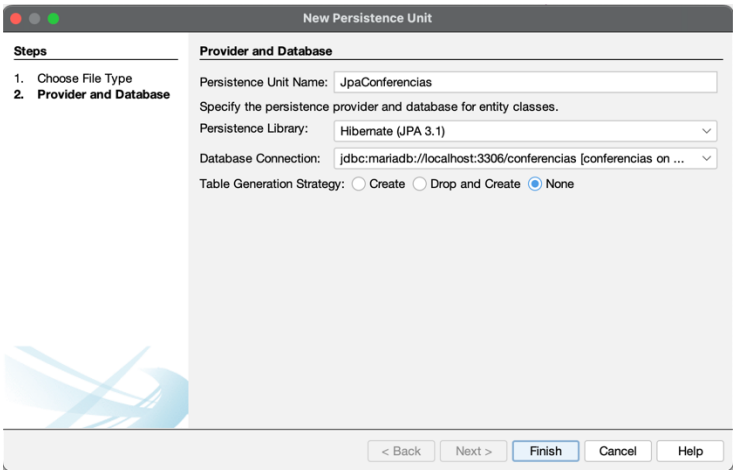
4. Crear la unidad de persistencia.

- Hacer clic derecho sobre el proyecto y seleccionar *New* → *Other...*
- En la categoría *Persistence*, seleccionar *Persistence Unit* y hacer clic en *Next*.

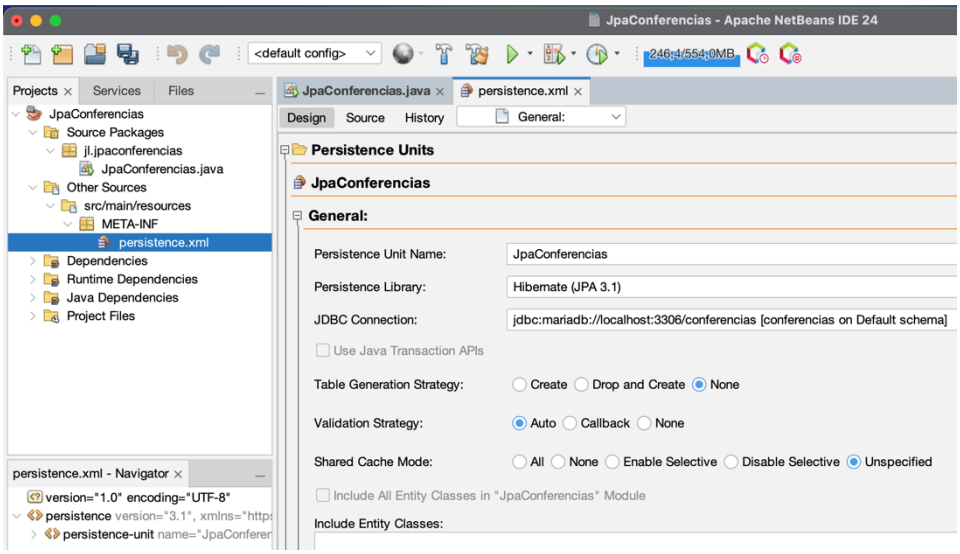




- Configurar la unidad de persistencia:
  - *Persistence Unit Name*: introducir un nombre para la unidad de persistencia (por ejemplo, `JpaConferencias`).
  - *Persistence Library*: seleccionar el proveedor *JPA* que se usará (por ejemplo, *Hibernate*).
  - *Database Connection*: seleccionar la conexión a la *BD* configurada en la sección Services.
  - *Table Generation Strategy*: elegir la estrategia de generación de tablas (por ejemplo, none).
    - *Create*: crea las tablas desde cero.
    - *Drop and Create*: elimina y vuelve a crear las tablas cada vez.
    - *None*: no realiza cambios en las tablas (recomendado si ya se tiene la *BD* configurada).



- Hacer clic en *Finish*. *NetBeans* generará automáticamente el archivo `persistence.xml` dentro de la carpeta `META-INF`.



Design Source History

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.1" xmlns="https://jakarta.ee/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
  <persistence-unit name="JpaConferencias" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost:3306/conferencias"/>
      <property name="jakarta.persistence.jdbc.user" value="conferencias"/>
      <property name="jakarta.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.password" value="conferencias"/>
      <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
    </properties>
  </persistence-unit>
</persistence>
```

Si no se tiene *Hibernate* configurado como una biblioteca reutilizable en *NetBeans*:

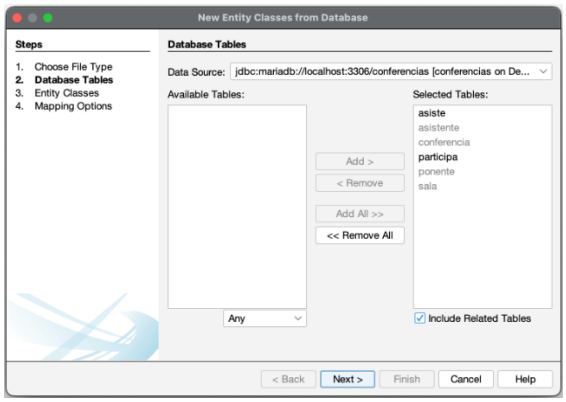
- Ir al menú *Tools* → *Libraries* en *NetBeans* o bien en la lista desplegable de selección de la *Persistence Library* seleccionar la opción *Manage Libraries…* (o bien seleccionar *New Persistence Library…*).
- Hacer clic en *New Library* y nombrarla (por ejemplo, *Hibernate*).
- Hacer clic en *Add JAR/Folder…* y seleccionar los archivos *JAR* descargados por *Maven* en el repositorio local (se pueden encontrar en `.m2/repository`).

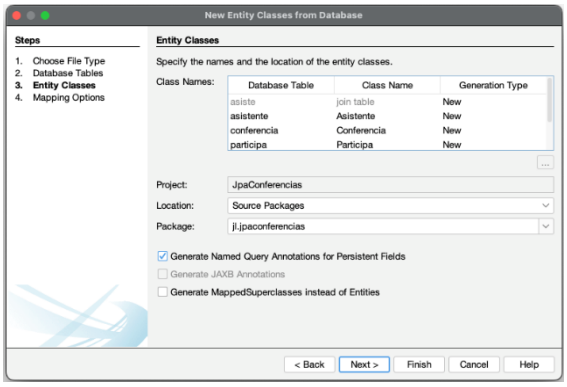
Desde el **editor gráfico** se puede **configurar el archivo `persistence.xml`**:

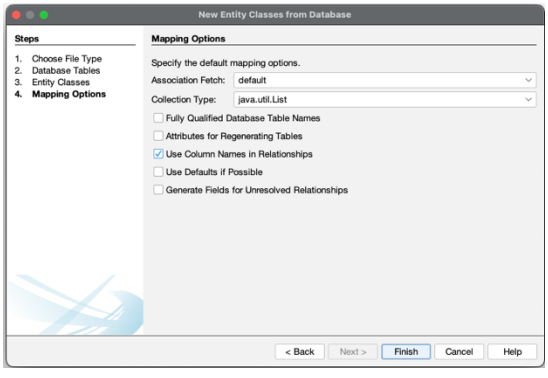
- Sección **General**:
  - Cambiar el nombre de la unidad de persistencia.
  - Seleccionar el proveedor de *JPA* (Hibernate, EclipseLink, etc.).
  - Configurar la conexión a la *BD*, como el usuario, contraseña y *URL JDBC*.
  - Definir las estrategias de generación de tablas (`create`, `update`, none).
  - Agregar o eliminar entidades persistentes en la unidad de persistencia.
- Sección **Properties**:
  - En esta pestaña se puede agregar o editar propiedades adicionales específicas del proveedor o de la configuración *JPA*.

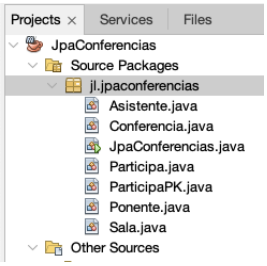
5. Generar las Entidades *JPA*.

- Hacer clic derecho sobre el proyecto *Maven* en la pestaña *Projects*.
- Seleccionar *New* → *Other...* → *Persistence* → *Entity Classes from Database*. Hacer clic en *Next >*.
- Seleccionar la conexión que se creó anteriormente.
- Añadir a *Selected Tables* las tablas que se desean mapear como entidades *JPA*.
- Hacer clic en *Next >*.
- Configurar el paquete de las entidades.
- Hacer clic en *Next >*.
- Seleccionar en *Collection Type* la opción `java.util.List`.





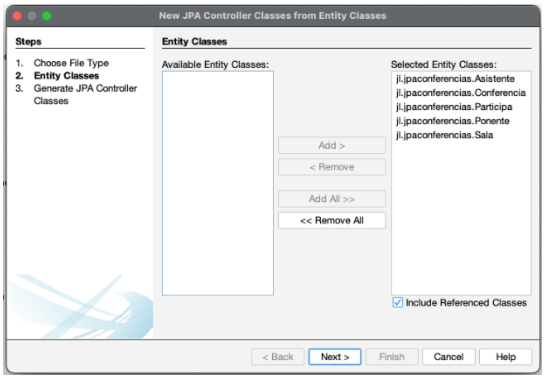




- Hacer clic en *Finish*.
- *NetBeans* generará las clases correspondientes en el paquete que se haya indicado.

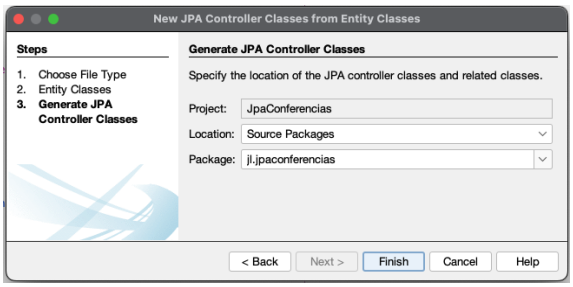
6. Crear los controladores *JPA*.

- Hacer clic derecho sobre el proyecto *Maven* en la pestaña *Projects*.
- Seleccionar *New* → *Other...* → *Persistence* → *JPA Controller Classes from Entity Classes*. Hacer clic en *Next >*.
- En la ventana del asistente se mostrarán las entidades disponibles

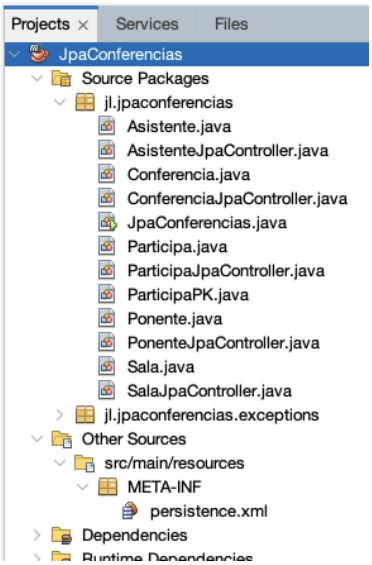


en el proyecto. Seleccionar las entidades para las que se desea generar los controladores.

- Haz clic en *Next >*.
- Especificar el paquete donde se generarán las clases controladoras.



- Hacer clic en *Finish* para completar la generación.
- *NetBeans* creará una clase controladora para cada entidad seleccionada. Estas clases incluyen métodos para realizar operaciones *CRUD* básicas



Los controladores generados se pueden personalizar:

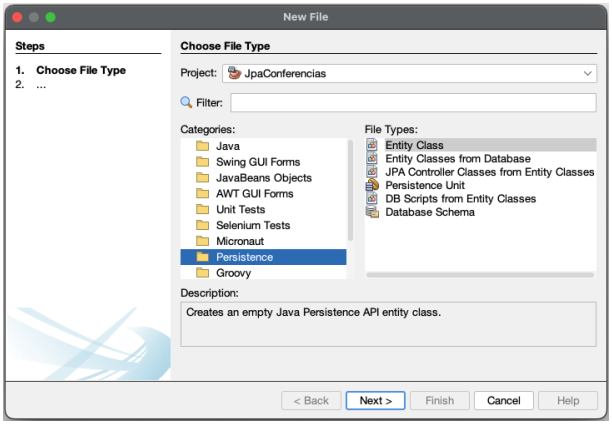
- ✓ Agregar métodos para consultas más complejas utilizando *EntityManager* y *JPQL*.
- ✓ Implementar validaciones adicionales antes de persistir o actualizar datos.

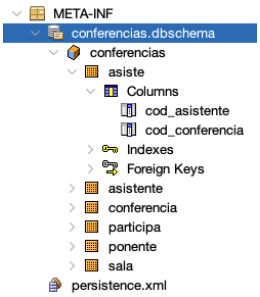
7. Implementar la lógica de negocio.

Una vez que se tienen las entidades y controladores *JPA* generados, el siguiente paso es implementar la lógica de negocio y las operaciones *CRUD* (crear, leer, actualizar y eliminar) en una clase principal o en las capas correspondientes del proyecto.

NetBeans ofrece **otras herramientas bajo el menú *Persistence*** que facilitan el trabajo con *JPA* y *BD*:

- ✓ ***Entity Class***: esta opción se utiliza para crear manualmente una clase de entidad *JPA* desde cero (*New* → *Other...* → *Persistence* → *Entity Class*). Usar esta opción si se necesita definir una entidad desde cero sin basarse en una tabla existente o si se está modelando una nueva tabla que aún no existe en la BD.
- ✓ ***DB Scripts from Entity Classes***: esta opción genera los scripts *SQL* necesarios para crear la estructura de la *BD* a partir de las clases de entidad existentes (*New* → *Other...* → *Persistence* → *DB Scripts from Entity Classes*). Usar esta opción si se está creando la *BD* desde cero y se necesita un archivo *SQL* para ejecutarlo en un *SGBD* como *MariaDB*, *PostgreSQL* o *SQLite*, o si se necesita distribuir el esquema de la *BD* junto con la aplicación.
- ✓ ***Database Schema***: esta opción permite analizar y mostrar visualmente el esquema de una *BD* desde las clases de entidad (*New* → *Other...* → *Persistence* → *Database Schema*). Usar esta opción para comprender mejor la estructura de las entidades y sus relaciones o para documentar el diseño de la *BD*.





Anexo II: `PersistenceUnitUtil`.

`PersistenceUnitUtil` es una **interfaz de *JPA* que proporciona métodos para obtener información sobre las entidades gestionadas sin realizar consultas a la *BD***. Los **métodos principales** son:

- ✓ **`isLoaded(Object entity)`**: comprueba si una entidad está completamente cargada.
- ✓ **`getIdentifier(Object entity)`**: obtiene el identificador (`ID`) de una entidad.
- ✓ **`isLoaded(Object entity, String attributeName)`**: comprueba si un atributo específico de una entidad

está cargado.

Se utiliza principalmente para comprobar el estado de las entidades o para obtener el identificador sin realizar consultas adicionales, optimizando el rendimiento y evitando acceso innecesario a la *BD*.

```java
import jakarta.persistence.*;
public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("mi_up");
        EntityManager em = emf.createEntityManager();
        // Obtener PersistenceUnitUtil
        PersistenceUnitUtil unitUtil = emf.getPersistenceUnitUtil();
        em.getTransaction().begin();
        Padre padre = new Padre("Juan");
        em.persist(padre);
        // Verificar si la entidad está cargada
        boolean isLoaded = unitUtil.isLoaded(padre);
        System.out.println("¿Está cargada la entidad 'padre'? " + isLoaded);
        // Obtener el identificador de la entidad
        Object idPadre = unitUtil.getIdentifier(padre);
        System.out.println("El ID del padre es: " + idPadre);
        em.getTransaction().commit(); // Confirmar la transacción
        em.close();
        emf.close();
    }
}
```

Anexo III: Lombok.

Lombok es una **biblioteca de *Java* que ayuda a reducir el código repetitivo, como getters, setters, constructores y métodos comunes** (toString, equals, hashCode), **mediante el uso de anotaciones**. Es ampliamente utilizada para simplificar el código en proyectos *Java* modernos.

Dependencia para *Maven*:

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.36</version>
</dependency>
```

Resumen de **anotaciones** de *Lombok*:

Anotación	Descripción
@Getter	Genera el método getter para los atributos.
@Setter	Genera el método setter para los atributos.
@ToString	Genera el método toString. Permite excluir campos con exclude.
@EqualsAndHashCode	Genera los métodos equals y hashCode.
@NoArgsConstructor	Genera un constructor vacío.
@AllArgsConstructor	Genera un constructor con todos los campos.
@RequiredArgsConstructor	Genera un constructor con los campos final o anotados con @NonNull.
@Data	Combina @Getter, @Setter, @ToString, @EqualsAndHashCode y constructores.
@Value	Similar a @Data, pero para clases inmutables (final).
@NonNull	Indica que un campo no puede ser null. Se usa en constructores o métodos.

***Lombok* se puede utilizar junto a *JPA* para simplificar el código en una aplicación que gestiona una entidad.**

Ejemplo: uso de las anotaciones de *Lombok* en una clase.

```java
import lombok.*;
@Data                       // Genera getters, setters, toString, equals y hashCode
@NoArgsConstructor          // Constructor sin argumentos
@AllArgsConstructor         // Constructor con todos los argumentos
public class Usuario {
    @NonNull                // Marca este campo como obligatorio en el constructor
    private Long id;
    private String nombreUsuario;
    private String correo;
    private String contrasena;
    @ToString.Exclude               // Excluye este campo del método toString
    @EqualsAndHashCode.Exclude      // Excluye este campo de equals y hashCode
    private String datosSensibles;
}

public class Principal {
    public static void main(String[] args) {
        // Crear instancia con el constructor vacío
        Usuario usuario1 = new Usuario();
        usuario1.setId(1L);
        usuario1.setNombreUsuario("JuanPerez");
        usuario1.setCorreo("juan.perez@ejemplo.com");
```

```
        usuario1.setContrasena("contrasenaSegura123");
        usuario1.setDatosSensibles("Información confidencial");

        // Crear instancia con el constructor completo
        Usuario usuario2 = new Usuario(2L, "MariaGomez", "maria.gomez@ejemplo.com", "contrasena456", "Más
datos sensibles");

        // Imprimir datos del usuario (excluye datosSensibles del toString)
        System.out.println(usuario1);
        System.out.println(usuario2);

        // Comparar dos objetos Usuario
        System.out.println("¿Son iguales? " + usuario1.equals(usuario2));

        // Modificar valores
        usuario1.setContrasena("nuevaContrasena");
        System.out.println("Contraseña actualizada: " + usuario1.getContrasena());
    }
}
```