

Tema 6. Herramientas de mapeo objeto relacional(ORM)

- 1. INTRODUCCIÓN**
- 2. PERSISTIR OBJETOS**
- 3. JAVA PERSISTENCE API (FRAMEWORK JPA)**
 - 3.1 Arquitectura JPA**
 - 3.2 ¿Cómo hace JPA para mapear objetos a una BD?**
 - 3.3 Principales componentes de JPA**
 - 3.3.1 Entity**
 - 3.3.2 Api EntityManager**
 - 3.3.3 Unidad de persistencia**
- 4 ANOTACIONES**
- 5 RELACIÓN ENTRE ENTIDADES**
 - 5.1 Asociaciones**
 - 5.2 Herencia**
- 6 OPERACIONES CON ENTIDADES**
 - 6.1 Persistir una entidad**
 - 6.2 Buscar una entidad**
 - 6.3 Actualizando entidades**
 - 6.4 Eliminando entidades**
 - 6.5 Sincronizando-Desincronizando**
 - 6.6 Métodos Callback**
- 7 JPQL(JAVA PERSISTENCE QUERY LANGUAGE)**
 - 7.1 Sentencias condicionales**
 - 7.2 Parámetros dinámicos**
 - 7.3 Obtener los resultados**
 - 7.4 Operaciones de actualización y borrado**
 - 7.5 Ejecución de sentencias con jpql**
 - 7.6 Ejecución de sentencias con parámetros dinámicos**
 - 7.7 Consultas con nombre: ESTATICAS**

1. INTRODUCCIÓN

En este tema se verá como acceder a una base de datos relacional utilizando un lenguaje orientado a objetos java.

Para esto es necesario una interfaz que traduzca la lógica de los objetos a la lógica relacional, esta interfaz se llama ORM (Object Relational Mapping).

Toda **aplicación** está formada por una interface de usuario y una lógica de negocio.

La interface de usuario es parte física que permite que el usuario introduzca datos o vea los datos que le devuelve la aplicación.

La lógica de negocio es el procesamiento de los datos y la entrega de los datos a través de la interface.



Arquitectura 3 capas

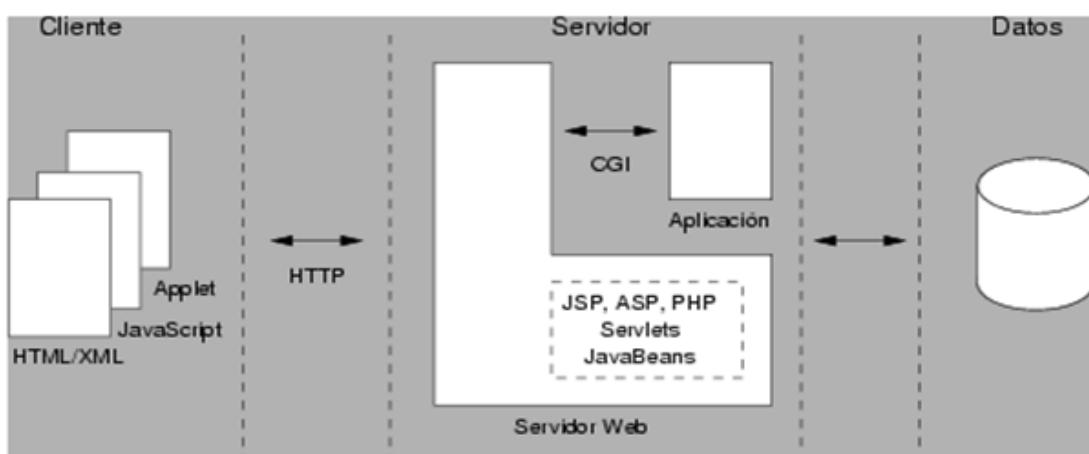


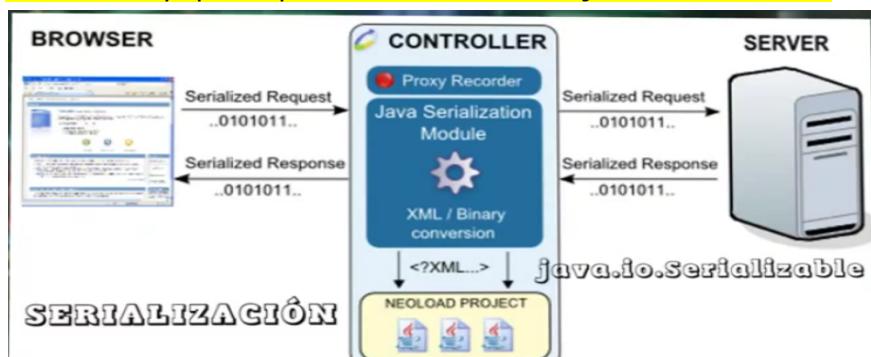
Figura: Esquema general de las tecnologías Web.

2. PERSISTIR OBJETOS

Java cuenta con varios modos:

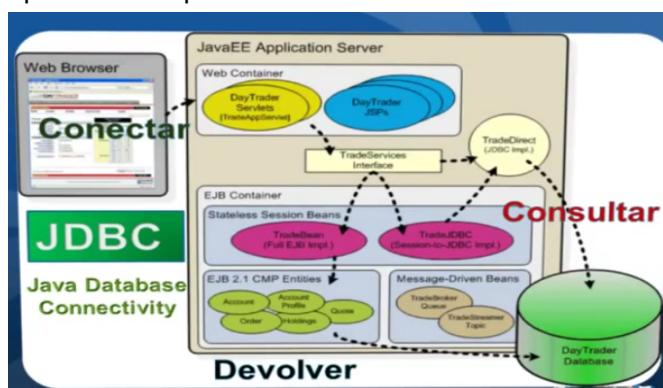
- a. **Serialización de objetos**
- b. **JDBC**
- c. **Herramientas ORM**

- a. **Serialización de objetos:** convertir los objetos en una secuencia de bits.
Para ello hay que implementar la interface `java.io.Serializable`



- b. **Jdbc (java database connectivity)**

Api estándar para acceder a una base de datos relacional

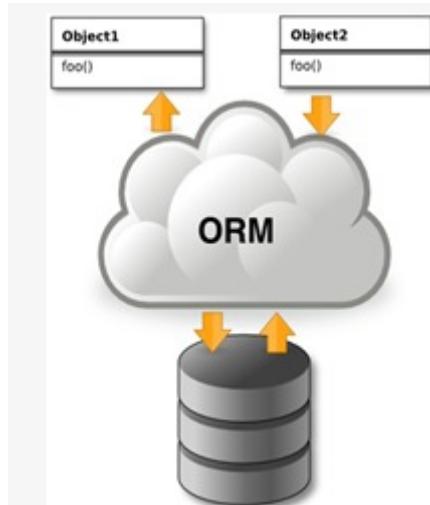


- c. **Herramientas ORM o de mapeo.**

El **mapeo objeto-relacional** (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo).

El objetivo es el delegar el acceso a bases de datos desde java en herramientas externas, en frameworks, lo que hacen es ofrecernos los datos

relacionales como si fueran una vista orientada a objetos y viceversa, los objetos como si fueran una vista de datos relacionales, para conseguirlos se usan las mapping tools.



Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

Existen varios framework que lo permiten:

Hibérnate, EclipseLink, JDO, Enterprise JavaBeans, EntityFramework.



Ventajas de las herramientas ORM

- No trabajar con filas de tablas
- Trabajar con las clases diseñadas en su modelo del dominio
- Permitir elegir la base de datos relacional con la que queremos trabajar
- Generar automáticamente el código SQL usando un mapeo objeto-relacional, el cual se especifica en un documento xml o en anotaciones
- Permitir crear, modificar, recuperar y eliminar objetos persistentes

Inconvenientes

- **Las aplicaciones son más lentas** debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

3. JAVA PERSISTENCE API (JPA)

JPA es la API de persistencia desarrollada para la plataforma Java EE. Es un **framework** del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE). Ha sido incluida en el estándar EJB3. La Persistencia en este contexto cubre tres áreas:

- La API en sí misma, definida en **javax.persistence.package**
- La Api Java Persistence Query Language (JPQL)
- Metadatos objeto/relacional

Ampliar con la documentación de oracle

<http://docs.oracle.com/javaee/5/api/javax/persistence/package-summary.html>

Package javax.persistence

The javax.persistence package contains the classes and interfaces that define the contracts between a persistence provider and the managed classes and the clients of the Java Persistence API.

See:

[Description](#)

Interface Summary

| | |
|--------------------------------------|--|
| EntityManager | Interface used to interact with the persistence context. |
| EntityManagerFactory | The EntityManagerFactory interface is used by the application to obtain an application-managed entity manager. |
| EntityTransaction | The EntityTransaction interface is used to control resource transactions on resource-local entity managers. |
| Query | Interface used to control query execution. |

Class Summary

| | |
|-----------------------------|--|
| Persistence | Bootstrap class that is used to obtain an EntityManagerFactory . |
|-----------------------------|--|

El objetivo de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional) y permitir usar objetos regulares conocidos como POJOs.

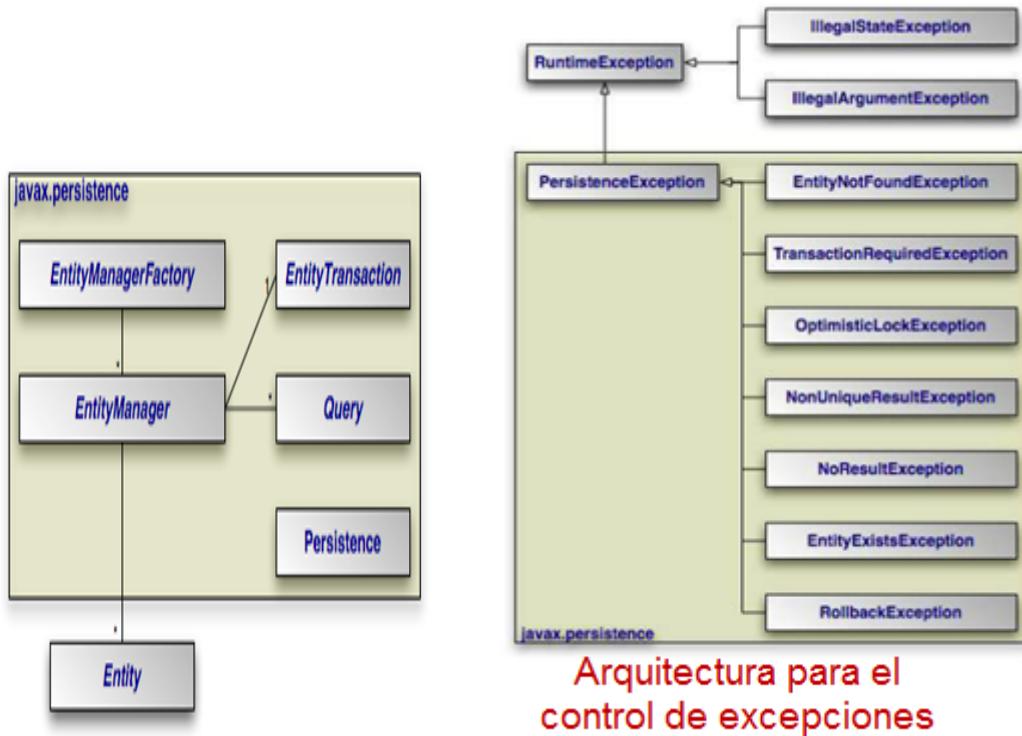


JPA es una abstracción sobre JDBC que nos permite realizar la correlación entre el sistema orientado a objetos de Java y el sistema relacional de la base de datos de forma sencilla, ya que realiza por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Todas las clases de ésta api están en el paquete **javax.persistence**.

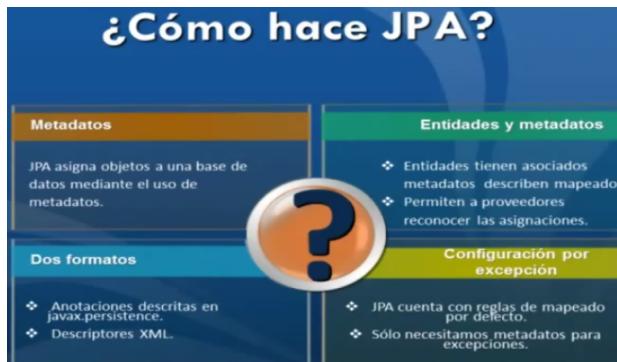
Esta conversión se llama **ORM** (Object Relation Mapping) y se configura a través de metadatos (xml o anotaciones).

JPA también permite seguir el sentido inverso, creando objetos a partir de las tablas de una base de datos, de forma transparente. A estos objetos se los llaman entidades o entities.

3.1 Arquitectura JPA



3.2 ¿Cómo hace JPA para mapear objetos a una BD?



Lo hace a través de metadatos, cada entidad tiene asignados los metadatos que describen el mapeado, la asignación, estos metadatos son los que habilitan al proveedor de persistencia para reconocer una entidad y habilitar el mapeado de la misma.

Pueden ser escritos de dos formatos:

- Como anotaciones descriptas en el paquete `javax.persistence`.
- Como descriptores XML, como fichero externo XML, muy útil cuando la configuración de la base de datos cambie dependiendo del entorno.

Jpa cuenta con ciertas reglas de mapeado por defecto, el nombre de la tabla es el mismo de la entidad, si estamos de acuerdo con estas reglas no vamos a tener que usar metadatos externos, ni anotaciones para indicar como va a ser el funcionamiento. Solamente si queremos personalizar el mapping podemos usar metadatos.

3.3 COMPONENTES DE LA ARQUITECTURA JPA



- a) El **ORM**, mecanismo para asignar objetos a datos almacenados en bases de datos relacionales.
- b) Una **Api EntityManager** administradora de la entidad, ejecuta operaciones en la base de datos relacional como son **crear**, **leer (read)**, **actualizar(update)** y **borrar(delete)**, se conocen como **CRUD**.
- c) En tercer lugar el **JPQL**, lenguaje persistente de consulta java, que permite recuperar datos con un lenguaje de consulta orientado a objetos.
- d) Uso de **listeners** para enlazar la lógica de negocio de un objeto persistente.

3.3.1 Entidades (ENTITY)

Las entities son simples pojos (plain old java objects) los pojo son clases simples que no dependen de un framework especial.



Si las entities son pojos, objetos java simples e independientes de cualquier plataforma, si son manejados por la EntityManager entonces tienen una identidad persistente y su estado se sincroniza con la base de datos, pero cuando no son manejados por una EntityManager, si los desconectamos de la EntityManager, pueden ser usados como cualquier otra clase de java, esto significa que las entities tienen un ciclo de vida



Cuando creamos una instancia de la Entity libro con el operador new, lo que estamos creando es un objeto que existe en memoria, pero de momento jpa no sabe nada sobre este objeto, por tanto es un objeto que puede terminar siendo recolectado como basura, cuando pasa a ser manejado por el EntityManager entonces es mapeado es asignado y sincronizado con la base de datos, en este caso con una tabla que llamamos libro. Si llamamos al método EntityManager.remove() lo que hacemos es borrar los datos, pero solo de la base de datos, porque el objeto java continua viviendo en la memoria hasta que es recolectado como basura.

Usaremos el término **entidad** en lugar de objetos, porque los objetos son instancias que solo viven en la memoria, mientras que las entidades son objetos que viven muy poco tiempo en la memoria y de modo persistente en la base de datos.

- Una entidad es una clase persistente.
- Una entidad representa una tabla en la base de datos.
- Cada instancia (objeto) de la entidad representa un registro en la base de datos.

Requerimientos para las Entidades

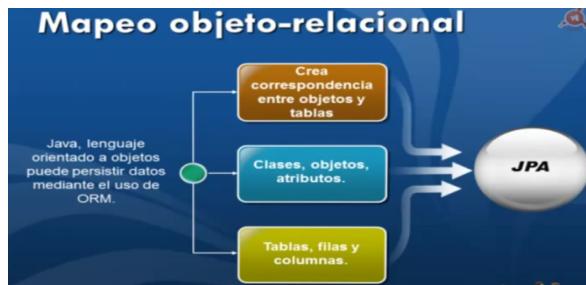
- La clase debe ser anotada con javax.persistence.Entity.
- La clase debe ser pública o protegida, y en ningún caso final.
- La clase debe tener al menos un constructor público o protegido sin argumentos.
- La clase puede implementar la interface Serializable.
- Una clase abstracta puede ser declarada como entidad.
- Todas las entidades deben tener una anotación de PK (Primary Key)
- Los atributos se deben declarar como privados.
- Los atributos no se deben declarar como final.
- Los atributos deben tener getters/setters públicos o protegidos.
- Los atributos no persistentes deben de ser anotados con javax.persistence.Transient.



Estas entidades cuentan con nuevas capacidades, pueden ser:

- Mapeadas (asignadas a una base de datos)
 - Pueden ser concretas o abstractas
 - Pueden soportar herencias, relaciones etc
- Una vez asignadas pueden ser manejadas por jpa, podemos persistir una entidad en la base de datos, consultarla, borrarla usando el lenguaje jpql.

Es el ORM, el mapeo objeto relacional el que nos permite manipular entidades.



Atributos persistentes

- Tipos primitivos.
- Matrices de tipos primitivos.
- `java.lang.String`
- `java.math.BigInteger` y `java.math.BigDecimal`
- `java.util.Date` y `java.util.Calendar`
- `java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`
- Clases serializables
- Clases tipo Enum
- Otras entidades
- Colecciones de todo lo anterior.
- `Java.util.Collection`, `java.util.List`, `java.util.Set`, `java.util.Map`

3.3.2 API EntityManager

- Administra la conexión con la base de datos.
- Administra la persistencia y la recuperación de entidades.
- Puede ser administrada por un contenedor JEE o por la aplicación.
- Puede administrar las transacciones o delegar en el contenedor JEE.
- Soporta la ejecución de consultas.

3.3.3 Unidad de Persistencia

Para que JPA pueda persistir esta entidad, necesita un archivo de configuración XML llamado '**persistence.xml**', el cual debe estar ubicado en el directorio **META-INF** de nuestra aplicación.

La agrupación de entidades en tu aplicación se llama *unidad de persistencia*.

El archivo persistence.xml contiene información necesaria para JPA, como el nombre de la unidad de persistencia, el tipo de transacciones que vamos a utilizar (concepto que veremos a continuación), las clases que deseamos que sean manejadas por el proveedor de persistencia, y los parámetros para conectar con nuestra base de datos.

El archivo persistence.xml tiene muchas funciones, pero la más importante tarea en un entorno de escritorio es la de listar todas las entidades en tu aplicación y nombrar la unidad de persistencia. Listar las clases entidad es necesario para la portabilidad en entornos Java SE.

persistence.xml

```

<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">
    <persistence-unit name="introduccionJPA" transaction-type="RESOURCE_LOCAL">
        <class>ejemplo.Cliente</class>
        <class>ejemplo.Orden</class>
        <class>ejemplo.Estudiante</class>
        <class>ejemplo.Curso</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.user" value="usuario"/>
            <property name="javax.persistence.jdbc.password" value="Pa$.w0rd"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://192.168.1.1/ejemplo"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

TRANSACCIONES

En el archivo de configuración anterior hemos incluido la siguiente línea:

```
<persistence-unit name="introduccionJPA" transaction-type="RESOURCE_LOCAL">
```

En ella se indica el nombre de la unidad de persistencia la cual pasaremos más adelante a EntityManager, para informarle de los parámetros de mapeo de un conjunto de entidades a gestionar, en nuestro caso solo una, y el tipo de transacción (RESOURCE_LOCAL) que utilizaremos al manejar este conjunto de entidades.

El concepto de transacción representa un contexto de ejecución dentro del cual podemos realizar varias operaciones como si fuera una sola, de manera que o todas ellas son realizadas satisfactoriamente, o el proceso se aborta en su totalidad (cualquier operación ya realizada se revertirá si ocurre un error a lo largo de la transacción). Esto es muy importante para garantizar la integridad de los datos que queremos persistir: imaginemos que estamos persistiendo una entidad que contiene una lista de entidades en su interior (una asociación uno-a-muchos). Si cualquiera de las entidades de esta lista no pudiera ser persistida por algún motivo, no deseamos que el resto de entidades de la lista, así como la entidad que las contiene, sean persistidas tampoco. Si permitiéramos que esto ocurriera, nuestros datos en la base de datos no reflejarían su estado real como objetos, y ni dichos datos ni nuestra aplicación serían fiables. JPA nos permite configurar en cada unidad de persistencia el tipo de transacción que queremos usar, que puede ser:

- Manejada por la aplicación (RESOURCE_LOCAL)
- Manejada por el contenedor (JTA)

En nuestros ejemplos utilizaremos transacciones manejadas por la aplicación. De esta manera, mantendremos los ejemplos simples (no necesitamos instalar y configurar un contenedor), además de alcanzar una comprensión mayor sobre el funcionamiento de las transacciones.

4. ANOTACIONES.

El API JPA usa anotaciones para definir clases que serán mapeadas a la base de datos. Las anotaciones **comienzan con @**.

@Entity

```
@Entity
Public class Empleado implements Serializable{}
```

En este ejemplo se creará una tabla para la entidad Empleado, por defecto el nombre de la tabla corresponde con el nombre de la clase.

@Table

Para definir los datos de una tabla de base de datos usamos la anotación @Table a nivel de declaración de la clase, así podemos definir el nombre de la tabla con la que se está mapeando la clase, el esquema, el catálogo. Si no se usa esta anotación y se usa sólo Entity, el nombre de la tabla será igual al nombre de la clase.

Para definir constraints de unicidad con la anotación **@UniqueConstraint**

Todas las entidades tienen que poseer una identidad que las diferencie del resto, por lo que deben contener una propiedad marcada con la anotación **@Id** (es aconsejable que dicha propiedad sea de un tipo que admita valores null, como Integer en lugar de int).

La identidad de una entidad va a ser gestionada por el proveedor de persistencia, así que será dicho proveedor quien le asigne un valor la primera vez que almacene la entidad en la base de datos. Para tal efecto, le añadimos a la propiedad de identidad la anotación **@GeneratedValue**.

@Column allows you to specify the name of the column in the database to which the attribute is to be persisted

Ejemplo: Entidad

```
import javax.persistence.*;
@Entity // obligatorio
@Table(name = "PERSONAS")
public class Persona {
    @Id // obligatorio
    @Column(name="ID_PERSONA")
    private int id;

    @Column(name="NOMBRE") // opcional
    private String nombre;

    public void setID();
    public String getID();

    public void setNombre();
    public String getNombre();
}
```

Existen distintos tipos de anotaciones para mapear propiedades. Esto depende del tipo de dato de las columnas de la tabla, si es una relación con otra tabla, etc.

Anotaciones de mapeo directo:

- **@Basic**
La anotación Basic es la más simple de todas, tiene dos atributos **fetch**, que define que estrategia se usa para traer los datos de la base de datos (FetchType.EAGER | FetchType.LAZY) y **optional** que toma valor booleano dependiendo de que el campo pueda o no ser nulo.
- **@Enumerated**
Se usa cuando se quiere guardar valores enumerados de constantes (Ordinales o de Texto). Tiene un solo atributo para configurar que es **value** (EnumType.ORDINAL | EnumType.STRING).
- **@Temporal**
Se usa para campos de tipo java.util.Date y java.util.Calendar.
- **@Lob**
Se usa esta anotación junto con un mapeo básico para especificar que una propiedad o campo persistente debe ser manejado como un Blob.
- **@Transient**
Por defecto JPA supone que todos los campos de una entidad son persistentes. Se usa esta anotación para indicar cuando una propiedad no es persistente.

Ciertas propiedades de una entidad pueden no representar su estado. Por ejemplo, imaginemos que tenemos una entidad que representa a una persona:

```
@Entity
public class Persona {
    @Id
    @GeneratedValue
    private Long id;
    private String nombre;
    private String apellidos
    private Date fechaNacimiento;
    private int edad;

    // getters y setters
}
```

Podemos considerar que la propiedad edad no representa el estado de Persona, ya que si no es actualizada cada cumpleaños, terminará conteniendo un valor erróneo. Ya que su valor puede ser calculado gracias a la propiedad fechaNacimiento, no vamos a almacenarlo en la base de datos, si no a calcular su valor en tiempo de ejecución cada vez que lo necesitemos. Para indicar al proveedor de persistencia que ignore una propiedad cuando realice el mapeo, usamos la anotación **@Transient**:

```
@Transient
private int edad;
```

Ahora, para obtener el valor de edad utilizamos su método getter:

```
public int getEdad() {
```

```
// calcular la edad y devolverla  
}
```

anotaciones de relaciones:

- **@OneToOne** (Uno a Uno)
Cuando una entidad A referencia a una Entidad B, y ningún otro A puede referenciar a B, decimos que tenemos una relación OneToOne.
- **@ManyToOne** (Muchos a Uno)
Cuando una entidad A referencia a una entidad B, y otros A pueden referenciar a este mismo B, decimos que tenemos una relación ManyToOne.
- **@OneToMany** (Uno a Muchos)
Cuando una entidad A refencia multiples entidades B, y no hay dos A que refencien el mismo B decimos que tenemos una relación OneToMany
- **@ManyToMany** (Muchos a Muchos)
Cuando muchas entidades A refencian a muchas entidades B, y otras entidades A pueden referenciar algunas de las mismas B decimos que tenemos una relación ManyToMany.
- **@MapKey**
- **@OrderBy**

5. RELACIÓN ENTRE ENTIDADES

Cuando queremos mapear colecciones de entidades, debemos usar asociaciones. Estas asociaciones pueden ser de dos tipos:

- Asociaciones unidireccionales
- Asociaciones bidireccionales

Las asociaciones unidireccionales reflejan un objeto que tiene una referencia a otro objeto (la información puede viajar en una dirección). Por el contrario, las asociaciones bidireccionales reflejan dos objetos que mantienen referencias al objeto contrario (la información puede viajar en dos direcciones). Además del concepto de dirección, existe otro concepto llamado cardinalidad, que determina cuantos objetos puede haber en cada extremo de la asociación.

ASOCIACIONES UNIDIRECCIONALES

Veamos un ejemplo de Unidireccionalidad en una relación de tipo uno-a-uno y en otra de tipo uno-a-muchos:

Uno a Uno @OneToOne

Cada entidad se relaciona con una sola instancia de otra entidad, donde ambas se referencian por la misma primary key.

```

@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Direccion direccion;

    // Getters y setters
}

@Entity
public class Direccion {
    @Id
    @GeneratedValue
    private Long id;
    private String calle;
    private String ciudad;
    private String pais;
    private Integer codigoPostal;
    // Getters y setters
}

```

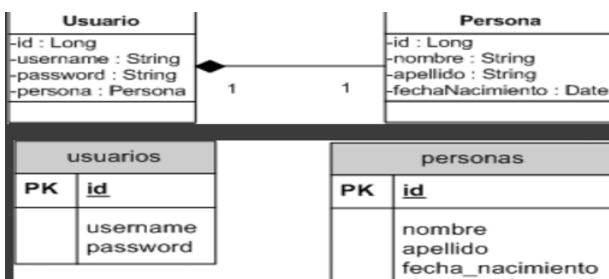
En el ejemplo anterior, cada entidad Cliente mantiene una referencia a una entidad Dirección. Esta relación es de tipo *uno-a-uno* (one-to-one) unidireccional (puesto que una entidad Cliente contiene una referencia a una entidad Dirección, relación que ha sido declarada mediante la anotación `@OneToOne`).

Cliente es el *dueño* de la relación de manera implícita, y por tanto cada entidad de este tipo contendrá por defecto una columna adicional en su tabla correspondiente de la base de datos (mediante la que podremos acceder al objeto Dirección). Esta columna es una *clave ajena* (foreign key)

Cuando JPA realice el mapeo de esta relación, cada entidad será almacenada en su propia tabla, añadiendo a la tabla donde se almacenan los clientes (la dueña de la relación) una columna con las claves ajenas necesarias para asociar cada fila con la fila correspondiente en la tabla donde se almacenan las direcciones. Recuerda que JPA utiliza configuración por defecto para realizar el mapeo, pero podemos personalizar este proceso definiendo el nombre de la columna que contendrá la clave ajena mediante la anotación `@JoinColumn`:

```
@OneToOne
@JoinColumn(name = "DIRECCION_FK")
private Direccion direccion;
```

Otro ejemplo)

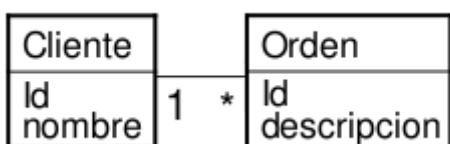
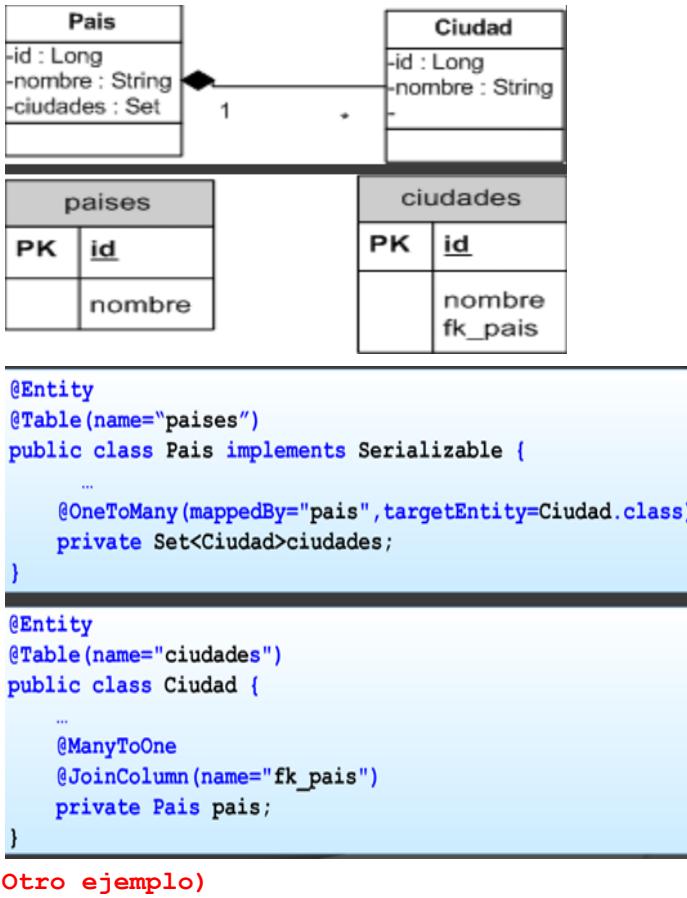


```
@Entity
@Table(name="usuarios")
public class Usuario implements Serializable {
    ...//username password id
    @OneToOne
    @JoinColumn(name="id")
    private Persona persona;
    //setter and getter
}

@Entity
@Table(name="personas")
public class Persona implements Serializable {
    ...//id nombre apellido fechaNacimiento
}
```

Uno a Muchos@OneToMany

La entidad, puede estar relacionada con varias instancias de otras entidades.



```

@Entity public class Cliente {
    ...
    @OneToMany(cascade=CascadeType.ALL, mappedBy="cliente")
        // lo voy mapear con cliente
    private Set<Orden> ordenes;// cliente tendrá una colección de órdenes
    ...
}

@Entity public class Orden {
    ...
    @ManyToOne
    @JoinColumn (name="ID_CLIENTE")
    private Cliente cliente; // mapeamos con cliente
    ...
}
  
```


ASOCIACIONES BIDIRECCIONALES

En las asociaciones bidireccionales, ambos extremos de la relación mantienen una referencia al extremo contrario. En este caso, el dueño de la relación debe ser especificado explícitamente, de manera que JPA pueda realizar el mapeo correctamente.

Veamos un ejemplo de bidireccionalidad en una relación de tipo **uno-a-uno**:

```

@Entity
public class Mujer {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Marido marido;

    // Getters y setters
}

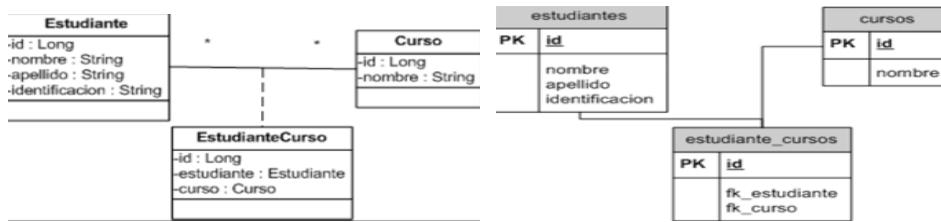
@Entity
public class Marido {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne(mappedBy = "marido")
    private Mujer mujer;
}

```

En el ejemplo anterior Mujer es la dueña de la relación: puesto que la relación es bidireccional, ambos lados de la relación deben estar anotados con @OneToOne, pero ahora uno de ellos debe indicar de manera explícita que la parte contraria es dueña de la relación. Esto lo hacemos añadiendo el atributo **mappedBy** en la anotación de asociación de la parte no-dueña. El valor de este atributo es el nombre de la propiedad asociada en la entidad que es dueña de la relación. El atributo mappedBy puede ser usado en relaciones de tipo @OneToOne, @OneToMany y @ManyToOne únicamente @ManyToOne, no permite el uso de este atributo.

Muchos a Muchos @ManyToMany

Varias instancias de una entidad pueden relacionarse con varias instancias de otra entidad.



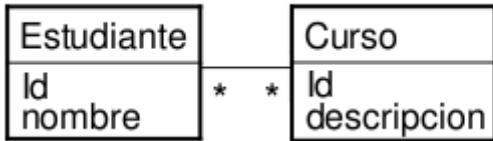
```

@Entity
@Table(name="estudiantes")
public class Estudiante implements Serializable {
    ...
    @ManyToMany
    @JoinTable(name="estudiante_cursos",
               joinColumns=@JoinColumn(name="fk_estudiante"),
               inverseJoinColumns=@JoinColumn(name="fk_curso"))
    private Set<Curso> cursos;
}
  
```

```

@Entity
@Table(name="cursos")
public class Curso implements Serializable{
    @Id
    @Column(name="id")
    private Long id;
    @Column(name="nombre")
    private String nombre;
    @Column(name="creditos")
    private Long creditos;
}
  
```

Otro ejemplo)



```

@Entity public class Estudiante {
    @ManyToMany(cascade=CCascadeType.ALL)
    @JoinTable(name="CURSO_ESTUDIANTE",
    joinColumns={@JoinColumn(name="ID_ESTUDIANTE", referencedColumnName="ID_E
    STUDIANTE")},
    inverseJoinColumns={@JoinColumn(name="ID_CURSO", referencedColumnName="ID
    _CURSO")})
    private List<Course> cursos;
        //la clase estudiante tendrá una colección de cursos y al
contrario
}
@Entity public class Curso {
    @ManyToMany(cascade=CCascadeType.ALL)
    @JoinTable(name="CURSO_ESTUDIANTE",

joinColumns=@JoinColumn(name="ID_CURSO", referencedColumnName="ID_CURSO"))

inverseJoinColumns=@JoinColumn(name="ID_ESTUDIANTE", referencedColumnName="ID_E
STUDIANTE"),
    private List<Student> estudiantes; //la clase curso tendrá una colección
de estudiantes
}
  
```

ORDENACIÓN DE ASOCIACIONES

Puedes ordenar los resultados devueltos por una asociación mediante la anotación `@OrderBy`:

```

@OneToMany
@OrderBy("nombrePropiedad asc")
private List pedidos;
  
```

El atributo de tipo String que hemos proporcionado a `@OrderBy` se compone de dos partes: el nombre de la propiedad sobre la que queremos que se realice la ordenación, y opcionalmente el sentido en que se realizara, que puede ser:

- Ascendente (añadiendo `asc` al final del atributo)
- Descendente (añadiendo `desc` al final del atributo)

Así mismo, podemos mantener ordenada la colección a la que hace referencia una asociación en la propia tabla de la base de datos; para ello, debemos usar la anotación `@OrderColumn`. Sin embargo, el impacto en el rendimiento de la base de datos que puede producir este comportamiento es algo que debes tener muy presente, ya que las tablas afectadas tendrán que reordenarse cada vez que se hayan cambios en ellas (en tablas de cierto tamaño, o en aquellas donde se

inserten o modifiquen registros con cierta frecuencia, es totalmente desaconsejable forzar una ordenación automática).

6. OPERACIONES CON ENTIDADES

Las operaciones que pueden llevar a cabo las entities se encuentran dentro de 4 categorías: persisting, updating, removing, loading que se corresponden con las operaciones de bases de datos inserting, updating, deleting y selecting, cada una de estas operaciones tiene un evento pre y otro post, excepto loading que solo tiene evento post.



EntityManager puede interceptar cuando tiene lugar cualquiera de estos eventos e invocar cualquier método de la lógica de negocio de la aplicación.

Contamos también con anotaciones @PrePersist y @PostPersist para enlazar la lógica de negocio con la entity cuando alguno de estos eventos se produce, estas anotaciones pueden ser configuradas como métodos entity también llamados métodos callback o clases externas también llamadas listeners.

Podemos pensar en los callbacks y los listeners como los tradicionales triggers que se usan en las bases de datos relacionales.

El concepto de persistencia implica el hecho de almacenar nuestras entidades (objetos Java de tipo POJO) en un sistema de almacenamiento, normalmente una base de datos relacional (tablas, filas, y columnas). Más allá del proceso de almacenar entidades en una base de datos, todo sistema de persistencia debe permitir recuperar, actualizar y eliminar dichas entidades. Estas cuatro operaciones se conocen como operaciones CRUD (Create, Read, Update, Delete - Crear, Leer, Actualizar, Borrar). JPA maneja todas las operaciones CRUD a través de la interface EntityManager.

Para que JPA pueda persistir esta entidad, necesita un archivo de configuración XML llamado 'persistence.xml', el cual debe estar ubicado en el directorio META-INF de nuestra aplicación.

ESTADO DE UNA ENTIDAD

Para JPA, una entidad puede estar en uno de los dos estados siguientes:

- Managed (gestionada)
- Detached (separada)

Cuando persistimos una entidad, automáticamente se convierte en una entidad *gestionada*. Todos los cambios que efectuemos sobre ella dentro del contexto de una transacción se verán reflejados también en la base de datos, de forma transparente para la aplicación.

El segundo estado en el que puede estar una entidad es *separado*, en el cual los cambios realizados en la entidad no están sincronizados con la base de datos. Una entidad se encuentra en estado separado antes de ser persistida por primera vez, y cuando tras haber estado gestionada es separada de su contexto de persistencia.

Persistiendo una entidad

EntityManager.persist(Entity);

```
Persona persona=new Persona();
persona.set(...);
entityManager.persist(persona);
```

ejemplo completo)

```
package ejemplo.jpa;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("introduccionJPA");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        Pelicula pelicula = new Pelicula();
        pelicula.setTitulo("Pelicula uno");
        pelicula.setDuracion(142);

        tx.begin();
        try {
            em.persist(pelicula);
            tx.commit();
        } catch(Exception e) {
            tx.rollback();
        }
    }
}
```

```

    em.close();
    emf.close();
}
}
}

```

En el ejemplo anterior hemos obtenido una instancia de EntityManager a través de la clase factoría EntityManagerFactory a la que le hemos pasado como argumento el nombre de la unidad de persistencia que vamos a utilizar (y que hemos configurado previamente en el archivo *persistence.xml*). A continuación hemos obtenido una transacción desde EntityManager, y hemos creado una instancia de la entidad Película. En las líneas de código siguientes, hemos iniciado la transacción, persistido la entidad, y confirmado la transacción mediante el método `commit()` de EntityManager.

En caso de que el proveedor de persistencia lance una excepción, la capturamos en el bloque catch, donde cancelamos la transacción mediante el método `rollback()` de EntityManager (en los próximos ejemplos, y para simplificar el código, vamos a eliminar el bloque try/catch, incluyendo el código que cancela la transacción en caso de error).

Cuando llamamos a `em.persist(pelicula)` la entidad es persistida en nuestra base de datos, película y queda gestionada por el proveedor de persistencia mientras dure la transacción. Por ello, cualquier cambio en su estado será sincronizado automáticamente y de forma transparente para la aplicación:

```

tx.begin();
em.persist(pelicula);
pelicula.setTitulo("otro titulo");
tx.commit();

```

En el ejemplo anterior hemos modificado el estado de la entidad película después de haber sido persistida, pero dentro de la misma transacción que realizó la persistencia, y por tanto la nueva información se sincronizará con la base de datos. Es importante tener presente que esta sincronización puede no ocurrir hasta que instamos a la transacción para completarse (`tx.commit()`) (el momento en que la sincronización se lleva a cabo depende enteramente de la implementación concreta del proveedor de persistencia que usemos). Sin embargo, podemos forzar a que cualquier cambio pendiente se guarde en la base de datos antes de terminar la transacción invocando el método `flush()` de EntityManager:

```

tx.begin();
em.persist(pelicula);
pelicula.setTitulo("otro titulo");
em.flush();      //fuerzo a que se guarde el cambio antes de terminar la transacción
// otras operaciones
tx.commit();

```

En el ejemplo anterior, los cambios efectuados sobre la entidad película (`setTitulo()`) serán persistidos, si no lo estuvieran ya, antes de finalizar la transacción. Por supuesto, si esta terminase fallando, todos los cambios efectuados durante la transacción, incluida la persistencia inicial de la entidad, serían revertidos (recuerda el concepto de transacción: todo o nada).

Así mismo, podemos realizar la sincronización en sentido inverso, actualizando una entidad con los datos que actualmente se encuentran en la base de datos. Esto es útil cuando persistimos una entidad, y tras haber cambiado su estado deseamos recuperar el estado persistido (desechando así los últimos cambios realizados sobre la entidad, que como sabemos podrían haber sido sincronizados con la base de datos). Esto podemos llevarlo a cabo mediante el método `refresh()` de EntityManager:

```
tx.begin();
em.persist(pelicula);
pelicula.setTitulo("otro titulo");
em.refresh(pelicula);
tx.commit();
```

En el ejemplo anterior, la llamada a `em.refresh()` deshace los cambios realizados en película en la línea anterior.

Otro método muy útil en EntityManager es `contains()`:

```
boolean gestionada = em.contains(pelicula);
// lógica de la aplicación
```

El método `contains()` devuelve true si el objeto Java que le pasamos como parámetro se encuentra en estado gestionado por el proveedor de persistencia, y false en caso contrario.

Buscando entidad

Leer una entidad previamente persistida en la base de datos para construir un objeto Java. Podemos llevar a cabo esto de dos maneras distintas:

- Obteniendo un objeto real (`find()`)
- Obteniendo una referencia a los datos persistidos (`getReference()`)

De la primera manera, los datos son leídos (por ejemplo, con el método `find()`) desde la base de datos y almacenados en una instancia de la entidad:

EntityManager.find(class,object);

El método find recibe como parámetros la clase de la entidad y valor de la clave primaria para buscar un sólo objeto de entidad. Si el objeto que estamos buscando no puede ser localizado por la clase EntityManager, entonces este método simplemente retornará null.

```
Pelicula p = em.find(Pelicula.class, id);

Long pk=1;
Persona persona=EntityManager.find(Persona.class,pk);
If(persona!=null){
    //Hacer algo con la persona...
}
```

La segunda manera de leer una entidad nos permite obtener una referencia a los datos almacenados en la base de datos, de manera que el estado de la entidad será leído de forma demorada (en el primer acceso a cada propiedad), no en el momento de la creación de la entidad:

EntityManager.getReference(class,object);

```
Pelicula p = em.getReference(Pelicula.class, id);
```

Este método es similar al método find(), toma como argumentos el nombre de la clase y el valor de la clave primaria, pero si no encuentra la entidad lanza una

EntityNotFoundException

En ambos casos, el valor de id debe ser el valor de id con el que fue persistida la entidad. Esta forma de lectura es muy limitada, de manera que luego veremos un sistema mucho más dinámico para buscar entidades persistidas: JPQL.

Actualizando entidades

Antes de explicar cómo actualizar una entidad, vamos a explicar cómo separar una entidad del contexto de persistencia. Podemos separar una o todas las entidades gestionadas actualmente por el contexto de persistencia mediante los métodos `detach()` y `clear()`, respectivamente. Una vez que una entidad se encuentra separada, esta deja de estar gestionada, y por tanto los cambios en su estado dejan de ser sincronizados con la base de datos. Si intentáramos llamar de nuevo al método `persist()` sobre una entidad separada, se lanzará una excepción (de hecho, la invocación de cualquier método que trabaje sobre entidades gestionadas lanzará una excepción cuando sea usado sobre entidades separadas).

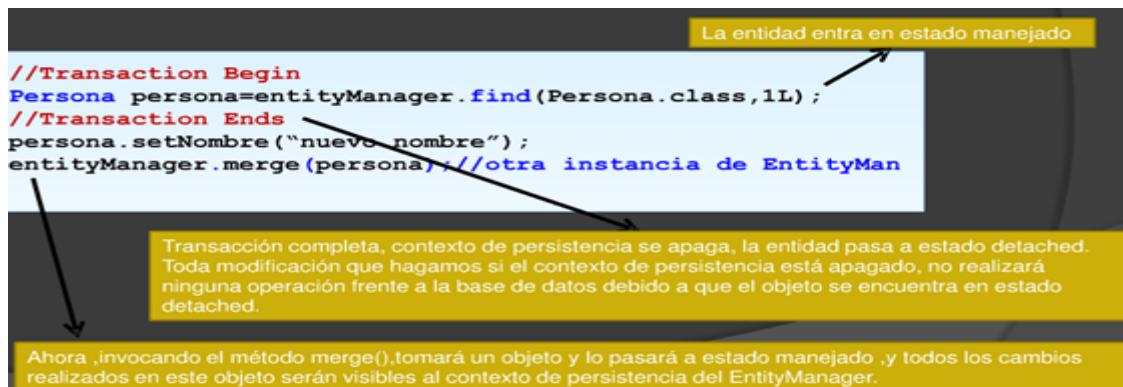
La pregunta que se nos plantearía en este momento es: ¿cómo podemos volver a tener nuestra entidad gestionada y sincronizada?. Mediante el método `merge()` de EntityManager:

```
tx.begin();
em.persist(pelicula);
tx.commit();
em.detach(pelicula);
// otras operaciones
em.merge(pelicula);
```

En el ejemplo anterior, la entidad película es separada del contexto de persistencia mediante la llamada al método `detach()`. La última línea, sin embargo, indica al proveedor de persistencia que vuelva a gestionar la entidad, y de manera adicional sincronice los cambios que se hayan podido realizar mientras la entidad estuvo separada (este pequeño milagro ocurre gracias al ID de la entidad).

EntityManager.merge(entidad)

Este método lo que hace es tomar una entidad en estado detached y asociarla al contexto de persistencia del entityManager, actualizando los datos del objeto en la base de datos.



Ejemplo

```

Persona public static Persona find(Long id){
    EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
    Persona persona=null;
    try{
        persona=em.find(Persona.class, id);
    }catch(Exception ex){
        System.out.println("upss!! ha ocurrido un error");
        ex.printStackTrace();
    }
    finally{
        em.close();
    }
    return persona;
}

public static void update(Persona persona){
    EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
    EntityTransaction tx=em.getTransaction();
    tx.begin();
    try{
        em.merge(persona);
        tx.commit();
        System.out.println("Actualizacion exitosa");
    }catch(Exception ex){
        tx.rollback();
        System.out.println("Ha ocurrido un error al actualizar");
        ex.printStackTrace();
    }
    finally{
        em.close();
    }
}

```

Main

```

public static void main(String[] args){
    Persona p=DAOPersona.find(3L);

    System.out.println("-----valores de base de datos-----");
    System.out.println(p);
    System.out.println("-----Cambios-----");

    p.setNombre("Ronald");
    p.setApellido("Cuello");

    DAOPersona.update(p);
}

```

Eliminando entidades

La última operación CRUD que nos queda por ver es la de eliminar una entidad. Cuando realizamos esta operación, la entidad es eliminada de la base de datos y separada del contexto de persistencia. Sin embargo, la entidad seguirá existiendo como objeto Java en nuestro código hasta que el ámbito de la variable termine, hasta que hipotéticamente sea puesta a null y el recolector de basura elimine la instancia de memoria, etc:

```

em.remove(pelicula);
pelicula.setTitulo("ya no soy una entidad, solo un objeto Java
normal");

```

Cuando existe una asociación uno-a-uno y uno-a-muchos entre dos entidades, y eliminas la entidad dueña de la relación, la/s entidad/es del otro lado de la relación no son eliminada/s de la base de datos (este es el comportamiento por defecto), pudiendo dejar así entidades persistidas sin ninguna entidad que haga referencia a ellas. Estas entidades se conocen como **entidades huérfanas**. Sin embargo, podemos configurar nuestras asociaciones para que eliminen de manera automática todas las entidades subordinadas de la relación:

```

@Entity
public class Pelicula {
    ...
@OneToOne(orphanRemoval = true)
private Descuento descuento;

// Getters y setters
}

```

En el ejemplo anterior, informamos al proveedor de persistencia que cuando elimine una entidad de tipo Pelicula, debe eliminar también de la base de datos la entidad Descuento asociada.

EntityManager.remove(entidad);

Para eliminar una entidad de la base de datos se debe invocar el método EntityManager.remove(EntityObject), para poder eliminar una entidad, esta debe ser una entidad manejada porque sino la operación fallará.

```

entityManager.getTransaction().begin();

Persona persona=entityManager.find(Persona.class,1L);
entityManager.remove(persona);

entityManager.getTransaction().commit();

```

Sincronizando entidades

Este método lo que hace es sincronizar los datos de una entidad y los hace persistentes en la base de datos

EntityManager.Flush()

Desincronizando entidades

Este método lo que hace es desincronizar los datos originales de la base datos y los vuelve a cargar en la entidad

EntityManager.Refresh(entity)



MÉTODOS CALLBACK

Una entidad puede estar en dos estados diferentes: gestionada y separada. Los métodos *callback* son métodos que se ejecutan cuando se producen ciertos eventos relacionados con el ciclo de vida de una entidad. Estos eventos se clasifican en cuatro categorías:

- (son como los triggers de bases de datos relacionales)
- Eventos de persistencia (métodos callback asociados anotados con **@PrePersist** y **@PostPersist**)
- Eventos de actualización (métodos callback asociados anotados con **@PreUpdate** y **@PostUpdate**)
- Eventos de borrado (métodos callback asociados anotados con **@PreRemove** y **@PostRemove**)
- Eventos de carga (método callback asociado anotado con **@PostLoad**)

Las anotaciones superiores están destinadas a marcar métodos dentro de la entidad, los cuales serán ejecutados cuando el evento correspondiente suceda:

```
@Entity
public class Pelicula {
    ...
    @PrePersist
    @PreUpdate
    private void validar() {
        // validar parametros antes de persistir/actualizar la entidad
    }
}
```

En el ejemplo anterior, hemos añadido a nuestra entidad un método que reaccionará antes dos eventos: el momento antes de realizarse la persistencia (@PrePersist), y el momento previo a la actualización (@PreUpdate). Dentro de él podemos realizar ciertas acciones necesarias antes de que se produzcan los dos citados eventos, como comprobar que el estado de la entidad (la información que contiene) es correcta, etc). Al escribir métodos callback, debemos seguir algunas reglas para que nuestro código funcione:

- Un método callback no puede ser declarado static ni final
- Cada anotación de ciclo de vida puede aparecer una y solo una vez en cada entidad
- Un método callback no puede lanzar excepción de tipo *checked*
- Un método callback pueden invocar métodos de las clases EntityManager y/o Query

7. JPQL (java query lenguaje)



JPQL usa la sintaxis objeto.col

Se pueden ejecutar consultas estáticas, dinámicas en tiempo de ejecución. En el caso de las estáticas se realizan también mediante el uso de metadatos, que tiene que ir en anotaciones (@NamedQuery) o en xml.

JPQL (JPA Query Language)

```

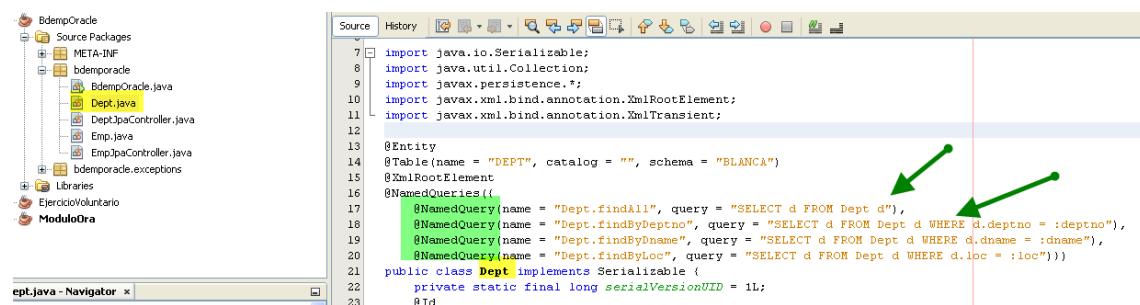
@Entity
@NamedQuery (name = «encontrarLibroPorTitulo»,
query = «SELECT I FROM Libro b WHERE I.titulo = «Curso de Java»
public class Libro {
@Id @GeneratedValue
private Long id;

@Column(nullable = false)
private String titulo;
private Float precio;
@Column(length = 2000)
private String descripcion;
private String isbn;
private Integer numPags;
private Boolean ilustraciones;
// Constructors, getters, setters
}

```

Ejemplo usamos la anotación @NamedQuery (en amarillo) para crear la consulta que llamamos encontrarLibroPorTitulo y a continuación la consulta con la sintaxis que se indicó antes

El método [EntityManager.createNamedQuery\(\)](#) es el usado para ejecutar la consulta y devolver las entity libro que coincidan con el criterio de búsqueda.



JPQL BÁSICO

Como vimos anteriormente al usar la interface EntityManager estamos limitados a realizar consultas en la base de datos proporcionando la identidad de la entidad que deseamos obtener, y solo podemos obtener una entidad por cada consulta que realicemos. JPQL nos permite realizar consultas en base a multitud de criterios (como por ejemplo el valor de una propiedad, o condiciones booleanas), y obtener más de un objeto por consulta. Veamos el ejemplo de sintaxis JPQL más simple posible:

```
SELECT p FROM Pelicula p
```

La sentencia anterior obtiene todas las instancias de la clase Película desde la base de datos. Las palabras SELECT y FROM tienen un significado similar a las sentencias homónimas del lenguaje SQL, indicando que se quiere seleccionar (SELECT) cierta información desde (FROM) cierto lugar. La segunda p es un alias para la clase Película, y ese alias es usado por la primera p (llamada expresión) para acceder a la clase (tabla) a la que hace referencia el alias, o a sus propiedades (columnas). El siguiente ejemplo nos ayudará a comprender esto mejor:

```
SELECT p.titulo FROM Pelicula p
```

El alias p nos permite utilizar la expresión p.título para obtener los títulos de todas las películas almacenadas en la base de datos. Las expresiones JPQL utilizan la notación de puntos, convirtiendo tediosas consultas en algo realmente simple:

```
SELECT c.propiedad.subPropiedad.subSubPropiedad FROM Clase c
```

JPQL también nos permite obtener resultados en base a más de una propiedad:

```
SELECT p.titulo, p.duracion FROM Pelicula p
```

Todas las sentencias anteriores (que más tarde veremos como ejecutar) devuelven, o un único valor, o un conjunto de ellos. Podemos eliminar los resultados duplicados mediante la cláusula DISTINCT:

```
SELECT DISTINCT p.titulo FROM Pelicula p
```

Así mismo, el resultado de una consulta puede ser el resultado de una función agregada aplicada a la expresión:

```
SELECT COUNT(p) FROM Pelicula p
```

COUNT() es una función agregada de JPQL, cuya misión es devolver el número de ocurrencias tras realizar una consulta. Por tanto, en el ejemplo anterior, el valor devuelto por la función agregada es el resultado de la sentencia al completo. Otras funciones agregadas son **AVG** para obtener la media aritmética, **MAX** para obtener el valor máximo, **MIN** para obtener el valor mínimo, y **SUM** para obtener la suma de todos los valores.

SENTENCIAS CONDICIONALES

Ahora que ya sabemos cómo realizar consultas básicas, vamos a introducir conceptos algo más complejos (pero aún simples). El primero de ellos es el de *consulta condicional*, el cual es aplicado añadiendo la cláusula WHERE en nuestra sentencia JPQL. Mediante una consulta condicional, restringimos los resultados devueltos por una consulta, en base a ciertos criterios lógicos (desde ahora la mayoría de los ejemplos constarán de varias líneas, pero ten presente que todos ellos representan una única sentencia JPQL, no varias):

```
SELECT p FROM Pelicula p
WHERE p.duracion < 120
```

La sentencia anterior obtiene todas las instancias de Pelicula almacenadas en la base de datos con una duración inferior a 120 minutos. Esto es llevado a cabo gracias al operador de comparación <. Las sentencias condicionales pueden más de una condición:

```
SELECT p FROM Pelicula p
WHERE p.duracion < 120 AND p.genero = 'Terror'
```

La sentencia anterior obtiene todas las instancias de Pelicula con una duración inferior a 120 minutos y cuya propiedad genero sea igual a Terror. Si en el caso anterior utilizamos un operador de comparación (<), en esta última sentencia hemos utilizado dos operadores de comparación (< y =), así como un operador lógico (AND). Los otros dos operadores lógicos disponibles en JPQL son OR y NOT. El primero de ellos, aplicado en el ejemplo anterior en lugar de AND, permitiría obtener todas películas con una duración inferior a 120 minutos, o las del género de Terror (solo una de las dos condiciones sería suficiente). El segundo de ellos, aplicado sobre un expresión, la niega:

```
SELECT p FROM Pelicula p
WHERE p.duracion < 120 AND NOT (p.genero = 'Terror')
```

En la sentencia anterior, se obtendrían todas las instancias de Pelicula con una duración menor a 120 minutos, y que *no* (NOT) son del género de terror. Veamos otro operador de comparación:

```
SELECT p FROM Pelicula p
WHERE p.duracion BETWEEN 90 AND 150
```

La sentencia anterior obtiene todas las instancias de Pelicula con una duración entre (BETWEEN) 90 y (AND) 150 minutos. BETWEEN puede ser convertido en NOT BETWEEN, en cuyo caso se obtendrían todas las películas que una duración que *no* (NOT) se encuentren dentro del margen (BETWEEN) 90-150 minutos. Otro operador de comparación muy útil es [NOT] LIKE (NOT es opcional, como en los ejemplos anteriores), el cual nos permite comparar una cadena de texto completa o solo definida en parte (esto último gracias al uso de comodines) con los valores de una propiedad almacenada en la base de datos. Veamos un ejemplo para comprenderlo mejor:

```
SELECT p FROM Pelicula p
WHERE p.titulo LIKE 'El%'
```

La sentencia anterior obtiene todas las instancias de Pelicula cuyo título sea como (LIKE) *El%* (el símbolo de porcentaje es un comodín que indica que en su lugar pueden haber entre cero y más caracteres). Resultados devueltos por esta consulta incluirían películas con un título como *El Caballero Oscuro*, *El Violinista en el Tejado*, o si existe, *El*. El otro comodín aceptado por LIKE es el carácter de barra baja (_), el cual representa un único carácter indefinido (ni cero caracteres ni más de uno; uno y solo uno).

PARÁMETROS DINÁMICOS

Podemos añadir parámetros dinámicamente a nuestras sentencias JPQL de dos formas: por posición y por nombre. La sentencia siguiente acepta un parámetro por posición (?1):

```
SELECT p FROM Pelicula p
WHERE p.titulo = ?1
```

Y la siguiente, acepta un parámetro por nombre (:titulo):

```
SELECT p FROM Pelicula p
WHERE p.titulo = :titulo
```

En el momento de realizar la consulta, deberemos pasar los valores con los que queremos que sean sustituidos los parámetros dinámicos que hemos definido.

ORDENAR LOS RESULTADOS

Cuando realizamos una consulta en la base de datos, podemos ordenar los resultados devueltos mediante la cláusula ORDER BY (ordenar por), la cual admite ordenamiento ascendente (mediante la cláusula ASC, comportamiento por defecto si omitimos el tipo de ordenamiento), o en orden descendiente (mediante la cláusula DESC):

```
SELECT p FROM Pelicula p
ORDER BY p.duracion DESC
```

La sentencia anterior podría tener una cláusula WHERE como las vistas en ejemplos

anteriores entre SELECT y ORDER BY, para restringir los resultados devueltos. Además, puedes incluir múltiples expresiones de ordenación en la misma sentencia:

```
SELECT p FROM Pelicula p
WHERE p.genero = 'Comedia'
ORDER BY p.duracion DESC, p.titulo ASC
```

En la sentencia anterior, hemos filtrado la selección de películas a las del género de comedia (mediante la cláusula WHERE), y hemos ordenado los resultados en base a dos criterios: por duración (DESC indica de mayor a menor duración en minutos), y entre las que tienen la misma duración, por título (ASC, que ya hemos dicho que es redundante por ser el comportamiento por defecto, indica de la A a la Z).

OPERACIONES DE ACTUALIZACIÓN

JPQL puede realizar operaciones de actualización en la base de datos mediante la sentencia UPDATE:

```
UPDATE Articulo a
SET a.descuento = 15
WHERE a.precio > 50
```

La sentencia anterior actualiza (UPDATE) todas las instancias de Articulo presentes en la base de datos cuyo precio (WHERE) sea mayor de 50, aplicándoles (SET) un descuento de 15.

OPERACIONES DE BORRADO

De forma muy similar a lo visto en la sección anterior, JPQL puede realizar operaciones de borrado en la base de datos mediante la sentencia DELETE:

```
DELETE FROM Pelicula p
WHERE p.duracion > 190
```

La sentencia anterior elimina (DELETE) todas las instancias de Pelicula cuya duración sea mayor de 190 minutos. Ni que decir tiene que las sentencias UPDATE y DELETE deben ser usadas con cierta precaución, sobre todo cuando trabajamos con información que se encuentra en producción.

EJECUCIÓN DE SENTENCIAS JPQL

El lenguaje JPQL es integrado a través de implementaciones de la interface Query. Dichas implementaciones se obtienen a través de EntityManager, mediante diversos métodos de factoría. De estos, los tres más usados son:

```
createQuery(String jpql)
createNamedQuery(String name)
createNativeQuery(String sql)
```

Empecemos viendo cómo crear un objeto Query y realizar una consulta a la base de datos:

```
package ejemplo.jpa;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("introduccionJPA");
        EntityManager em = emf.createEntityManager();

        String jpql = "SELECT p FROM Pelicula p";
        Query query = em.createQuery(jpql);
        List<Pelicula> resultados = query.getResultList();
        for(Pelicula p : resultados) {
            // ...
        }

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

En el [ejemplo anterior](#), obtenemos una implementación de Query mediante el método `createQuery(String)` de `EntityManager`, al cual le pasamos una sentencia JPQL en forma de cadena de texto. Con el objeto `Query` ya inicializado, podemos realizar la consulta a la base de datos llamando a su método `getResultList()`, el cual devuelve un objeto `List` con todas las entidades devueltas por la sentencia JPQL. Esta sentencia es una sentencia dinámica, ya que es generada cada vez que se ejecuta. De manera adicional, el ejemplo nos muestra que al usar una colección parametrizada (`List<Pelicula>`) nos evitamos tener que hacer ningún tipo de casting al manejar las entidades (al fin y al cabo JPA está devolviendo entidades de clases concretas, así que podemos aprovechar esta circunstancia usando colecciones genéricas).

EJECUCIÓN DE SENTENCIAS CON PARÁMETROS DINÁMICOS

```
String jpql = "SELECT p FROM Pelicula p WHERE p.duracion > ?1 AND
p.genero = ?2"
Query query = em.createQuery(jpql);
query.setParameter(1, 180);
query.setParameter(2, "Accion");
List<Pelicula> resultados = query.getResultList();
```

En el ejemplo anterior, hemos insertado dinámicamente (mediante el método setParameter()) los valores deseados para las expresiones p.duración y p.género, que en la sentencia JPQL original se corresponden con los parámetros por posición ?1 y ?2, respectivamente. El primer argumento que pasamos a setParameter() indica que parámetro por posición deseamos sustituir por el valor del segundo argumento. Si el valor que pasamos como segundo argumento no se corresponde con el valor esperado (por ejemplo, al pasar un una cadena de texto donde se espera un valor numérico), la aplicación lanzará una excepción de tipo IllegalArgumentException. Esto también ocurrirá si intentamos dar un valor a un parámetro dinámico inexistente (como query.setParameter(3, "Valor") en nuestro ejemplo anterior).

El otro tipo de parámetro dinámico (por nombre)

```
String jpql = "SELECT p FROM Pelicula p WHERE p.duracion > :duracion
AND p.genero = :genero"
Query query = em.createQuery(jpql);
query.setParameter("duracion", 180);
query.setParameter("genero", "Accion");
List<Pelicula> resultados = query.getResultList();
```

En el ejemplo anterior, en lugar de utilizar ?1 y ?2 en la sentencia JPQL, hemos utilizado :duración y :genero como parámetros dinámicos. Para poder darle valor a estos parámetros en el momento de realizar la consulta, setParameter() provee una versión cuyo primer argumento acepta un valor de tipo String con el que poder identificar el parámetro dinámico (query.setParameter("duracion", 180)). Que versión usar depende de preferencias personales, pues ambos cumplen exactamente la misma misión; sin embargo, es evidente que los parámetros dinámicos por nombre son más fáciles de identificar, entender.

CONSULTAS CON NOMBRE (ESTÁTICAS) @NamedQuery

Las consultas con nombre son diferentes de las sentencias dinámicas que hemos visto hasta ahora en el sentido de que una vez definidas, no pueden ser modificadas: son leídas y transformadas en sentencias SQL cuando el programa arranca por primera vez, en lugar de cada vez que son ejecutadas. Este comportamiento estático las hace más eficientes, y por tanto ofrecen un mejor rendimiento. Las consultas con nombre son definidas mediante metadatos (recuerda que los metadatos se definen mediante anotaciones o configuración XML)

```
@Entity
@NamedQuery(name="buscarTodos", query="SELECT p FROM Pelicula p")
public class Pelicula { ... }
```

El ejemplo anterior define una consulta con nombre a través de la anotación **@NamedQuery**. Esta anotación necesita dos atributos: name (que define el nombre de la consulta), y query (que define la sentencia JPQL a ejecutar). El nombre de la consulta debe ser único dentro de su unidad de persistencia, y por tanto no pueden existir dos entidades dentro de la citada unidad de persistencia que definan consultas estáticas con el mismo nombre. Para evitar que podamos modificar por error la sentencia, es una buena idea utilizar una constante definida dentro de la propia entidad, y usarla como nombre de la consulta:

```
@Entity
@NamedQuery(name=Pelicula.BUSCAR_TODOS, query="SELECT p FROM Pelicula
p")
public class Pelicula {
    public static final String BUSCAR_TODOS = "Pelicula.buscarTodos";
    ...
}
```

De manera adicional, esto nos permite crear una consulta con el mismo nombre en múltiples entidades (como BUSCAR_TODOS), pues ahora el nombre de la entidad se encuentra incluido en el nombre de la consulta, y por tanto seguimos sin violar la regla de *nombres únicos para todas las consultas dentro de la misma unidad de persistencia*. Consultas similares con nombres similares en entidades distintas harán nuestro código más fácil de escribir y mantener.

Una vez definida la consulta con nombre, podemos crear el objeto Query necesario mediante el método `createNamedQuery()`:

```
Query query = em.createNamedQuery(Pelicula.BUSCAR_TODOS);
List<Pelicula> resultados = query.getResultList();
```

`createNamedQuery()` requiere un parámetro de tipo `String` que contenga el nombre de la consulta (el cual hemos definido a través del parámetro `name` de `@NamedQuery`). Una vez creado el objeto `Query`, podemos trabajar con él de la manera habitual para obtener los resultados.

Si miramos el código generado en los ejercicios de clase por ejemplo, `persistenciaBdemp`, editamos la clase `Dept.java` veremos que ha generado automáticamente unas consultas básicas. Lo mismo para `Emp.java`.

Se pueden añadir más modificando el código.

```

package persistenciabdemp;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlTransient;

@Entity
@Table(name = "DEPT")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Dept.findAll", query = "SELECT d FROM Dept d"),
    @NamedQuery(name = "Dept.findByDeptno", query = "SELECT d FROM Dept d WHERE d.deptno = :deptno"),
    @NamedQuery(name = "Dept.findByDname", query = "SELECT d FROM Dept d WHERE d.dname = :dname"),
    @NamedQuery(name = "Dept.findByLoc", query = "SELECT d FROM Dept d WHERE d.loc = :loc"))
public class Dept implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "DEPTNO")
    private Integer deptno;
    @Column(name = "DNAME")
    private String dname;
    @Column(name = "LOC")
    private String loc;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "deptno")
    private Collection<Emp> empCollection;

    public Dept() {
}

```

Otros ejemplos

```

em.flush(); // fuerza a que los cambios pendientes se guarden en la base de datos
Cliente c = em.find(Cliente.class, id); // recupera cliente de la base de datos por ID
em.refresh(cliente); // recarga de la base de datos y sobreescribe en memoria
em.remove(cliente); // borra de la base de datos
//JPQL: una query
String jpql = "SELECT c FROM Cliente c";
Query query = em.createQuery(jpql);
List<Cliente> clientes = query.getResultList();
for(Cliente c : clientes) {
    // ...
}
//JPQL: query con parámetros (I)
String jpql = "SELECT c FROM Cliente c WHERE c.nombre = ?1 AND c.id > ?2";
Query query = em.createQuery(jpql);
query.setParameter(1, "Pepito");
query.setParameter(2, 468);
List<Cliente> clientes = query.getResultList();
for(Cliente c : clientes) {
    // ...
}
//JPQL: query con parámetros por nombre (II)
String jpql = "SELECT c FROM Cliente c WHERE c.nombre = :nombre AND c.id > :id";

```

```
Query query = em.createQuery(jpql);
query.setParameter("nombre", "Pepito");
query.setParameter("id", 468);
List<Cliente> clientes = query.getResultList();
for(Cliente c : clientes) {
    // ...
}
```