Unidad Persistencia mediante ORM : Maven, JPA, Hibernate

Índice

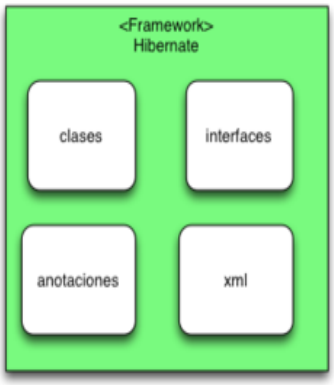
1. Introducción

1.1. Maven, JPA,Hibernate

- Maven: Maven es una herramienta open-source, que se creó en 2001 con el objetivo de simplificar los procesos de build (compilar y generar ejecutables a partir del código fuente).
- JPA (API de Persistencia en Java) es el estándar en Java que define una abstracción que nos permite realizar la integración entre un sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos.
  JPA realiza por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Esta conversión se llama ORM (Mapeo Relacional de Objetos) y puede configurarse a través de metadatos (XML) o anotaciones.

  JPA establece un interface común que es implementada por un "JPA Proveedor" concreto. De modo que es el Proveedor JPA el que realiza el trabajo. Entre los proveedores JPA más conocidos se encuentran Hibernate, Eclipse Link, TopLink, OpenJPA.

- Hibernate es un framework que gestiona (o implementa) la capa de persistencia a traves de ficheros xml o anotaciones.



1.2. ¿Qué es JPA?

Java Persistence API, más conocida por sus siglas 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**).

Persistencia en este contexto cubre tres áreas:

- La **API** en sí misma, definida en el paquete javax.persistence o jakarta.persistence según JDK de Java.
- El lenguaje de consulta **Java Persistence Query Language** (JPQL).
- Metadatos objeto/relacional.

El objetivo que persigue el diseño 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**), como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como **POJOs**).

En Java solucionamos problemas de negocio a través de objetos, los cuales tienen estado y comportamiento. Sin embargo, las bases de datos relacionales almacenan la información mediante tablas, filas, y columnas, de manera que para almacenar un objeto hay que realizar una correlación entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos.

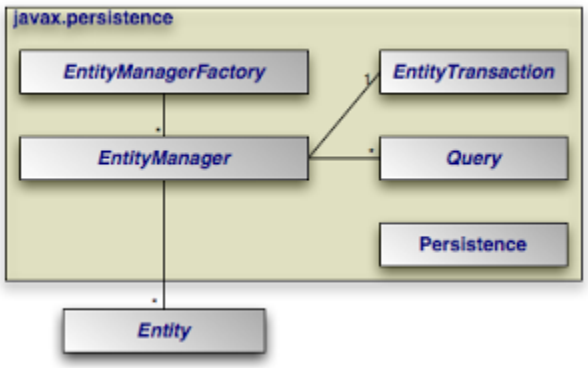
JPA (Java Persistence API, API de Persistencia en Java) es una abstracción sobre JDBC que nos permite realizar dicha correlación de forma sencilla, realizando por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Esta conversión se llama ORM (Object Relational Mapping - Mapeo Relacional de Objetos), y puede configurarse a través de metadatos (mediante xml o anotaciones).

Por supuesto, JPA también nos permite seguir el sentido inverso, creando objetos a partir de las tablas de una base de datos, y también de forma transparente. A estos objetos los llamaremos desde ahora *entidades* (entities).

JPA establece una interface común que es implementada por un proveedor de persistencia de nuestra elección (como Hibernate, Eclipse Link, etc), de manera que podemos elegir en cualquier momento el proveedor que más se adecue a nuestras necesidades. Así, es el proveedor quién realiza el trabajo, pero siempre funcionando bajo la API de JPA.

1.3. Arquitectura JPA

La arquitectura de JPA está diseñada para gestionar Entidades y las relaciones que hay entre ellas. A continuación detallamos los principales componentes de la arquitectura



Entity: Una entidad de persistencia (entity) es una clase de Java ligera, cuyo estado es persistido de manera asociada a una tabla en una base de datos relacional. Las instancias de estas entidades corresponden a un registro (conjunto de datos representados en una fila) en la tabla. Normalmente las entidades están relacionadas a otras entidades, y estas relaciones son expresadas a través de meta datos objeto/relacional. Los meta datos del objeto/relacional pueden ser especificados directamente en el fichero de la clase, usando las anotaciones de Java (annotations), o en un documento descriptivo XML, el cual es distribuido junto con la aplicación.

Persistence: Clase con métodos estáticos que nos permiten obtener instancias de EntityManagerFactory

EntityManagerFactory: Es una factoría de EntityManager. Se encarga crear y gestionar múltiples instancias de EntityManager

EntityManager: Es una interface que gestiona las operaciones de persistencia de las Entidades. A su vez trabaja como factoría de Query

Query: Es una interface para obtener la relación de objetos que cumplen un criterio

EntityTransaction: Agrupa las operaciones realizadas sobre un EntityManager en una única transacción de Base de Datos

1.4. Una entidad simple

Este es el ejemplo más simple de entidad:

```
@Entity
public class Pelicula {
   @Id
   @GeneratedValue
   private Long id;
   private String titulo;
   private int duracion;

   // Getters y Setters
}
```

Las entidades suelen ser POJOs.

* Un **POJO** (acrónimo de *Plain Old Java Object*) es una sigla creada por Martin Fowler, Rebecca Parsons y Josh MacKenzie en septiembre de 2000 y utilizada por programadores Java para enfatizar el uso de clases simples y que no dependen de un framework en especial.

La clase `Pelicula` se ha anotado con `@Entity`, lo cual informa al proveedor de persistencia que las instancias de esta entidad corresponden a un registro (conjunto de datos representados en una fila) en la tabla.

Para ser válida, toda entidad debe:

- Proporcionar un constructor por defecto (ya sea de forma implícita o explícita)
- Ser una clase de primer nivel (no interna)
- No ser final
- Implementar la interface `java.io.Serializable` si va a ser accedida remotamente

La configuración de mapeo puede ser especificada mediante un archivo de configuración XML, o mediante anotaciones (usaremos este último). A esta configuración del mapeo le llamamos **metadatos**.

1.5. Identidad

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`).

Si se quiere que la identidad de una entidad sea 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), le añadimos a la propiedad de identidad la anotación `@GeneratedValue`.

1.6. Configuración por defecto

JPA aplica a las entidades que maneja, una configuración por defecto, de manera que una entidad es funcional con una mínima cantidad de información (las anotaciones `@Entity` y `@Id` en nuestro caso).

Con esta configuración por defecto, todas las entidades del tipo `Pelicula` serán mapeadas a una tabla de la base de datos llamada PELICULA, y cada una de sus propiedades será mapeada a una columna con el mismo nombre (la propiedad `id` será mapeada en la columna ID, la propiedad `titulo` será mapeada en la columna TITULO, etc).

Sin embargo, no siempre es posible ceñirse a los valores de la configuración por defecto: imaginemos que tenemos que trabajar con una base de datos heredada, con nombres de tablas y filas ya definidos. En ese caso, podemos configurar el mapeo de manera que se ajuste a dichas tablas y filas:

```
@Entity
@Table(name = "TABLA_PELICULAS")
public class Pelicula {
   @Id
   @GeneratedValue
   @Column(name = "ID_PELICULA")
   private Long id;

   // ...
}
```

La anotación `@Table` nos permite configurar el nombre de la tabla donde queremos almacenar la entidad mediante el atributo `name`.
Así mismo, mediante la anotación `@Column` podemos configurar el nombre de la columna donde se almacenará una propiedad, si dicha propiedad puede contener un valor `null`, etc.

1.7. Lectura temprana y lectura demorada

Cuando leemos una entidad desde la base de datos, ciertas propiedades pueden no ser necesarias en el momento de la creación del objeto.

JPA nos permite leer una propiedad desde la base de datos la primera vez que un cliente intenta leer su valor (lectura demorada), en lugar de leerla cuando la entidad que la contiene es creada (lectura temprana).

De esta manera, si la propiedad nunca es accedida, nos evitamos el coste de crearla.

Esto puede ser útil si la propiedad contiene un objeto de gran tamaño:

```
@Basic(fetch = FetchType.LAZY)
private Imagen portada;
```

La propiedad `portada` es un objeto que representa una imagen de la portada de una película (un objeto de gran tamaño). Puesto que el coste de crear este objeto al leer la entidad `Pelicula` es muy alto, lo hemos marcado como una propiedad de lectura demorada mediante la anotación `@Basic(fetch = FetchType.LAZY)`.

El comportamiento por defecto de la mayoría de tipos Java es el contrario (lectura temprana). Este comportamiento, a pesar de ser implícito, puede ser declarado explícitamente de la siguiente manera:

```
@Basic(fetch = FetchType.EAGER)
private Imagen portada;
```

Sólo los objetos de gran tamaño y ciertos tipos de asociación deben ser leidos de forma demorada. Si, por ejemplo, marcamos todas las propiedades de tipo `int`, `String` o `Date` de una entidad con lectura demorada, cada vez que accedamos por primera vez a cada una de ellas el proveedor de persistencia tendrá que hacer una llamada a la base de datos para leerla. Esto, evidentemente, va a provocar que se efectúen multitud de llamadas a la base de datos cuando con solo una (al crear la entidad en memoria) podrían haberse leído todas con apenas coste.

Por tanto, es importante tener presente que la manera de configurar el tipo de lectura de una propiedad puede afectar enormemente al rendimiento de nuestra aplicación.

Por otro lado, la anotación `@Basic` solo puede ser aplicada a tipos primitivos, sus correspondientes clases wrapper, `BigDecimal`, `BigInteger`, `Date`, arrays, algunos tipos del paquete `java.sql`, enums, y cualquier tipo que implemente la interface `Serializable`.
Además del atributo `fetch`, la anotación `@Basic` admite otro atributo, `optional`, el cual permite definir si la propiedad sobre la que se está aplicando la anotación puede contener el valor `null` (esto es debido a que en bases de datos relacionales, algunas columnas pueden definir una restricción de tipo *non null*, la cual impide que se inserte un valor `null`; por tanto, con `@Basic(optional=false)` nos ajustaríamos a la citada restricción).

1.8. Tipos enumerados

JPA puede mapear los *tipos enumerados* (enum) mediante la anotación `Enumerated`:

```
@Enumerated
private Genero genero;
```

La configuración por defecto de JPA mapeará cada valor ordinal de un tipo enumerado a una columna de tipo numérico en la base de datos. Por ejemplo, siguiendo el ejemplo anterior, podemos crear un tipo `enum` que describa el género de una película:

```
public enum Genero {
   TERROR,
   DRAMA,
   COMEDIA,
```

```
   ACCION,
}
```

Si la propiedad `genero` del primer ejemplo tiene el valor `Genero.COMEDIA`, en la columna correspondiente de la base de datos de insertará el valor `2` (que es el valor ordinal de `Genero.COMEDIA`). Sin embargo, si en el futuro introducimos un nuevo tipo de genero en una posición intermedia, o reordenamos las posiciones de los géneros, nuestra base de datos contendrá valores erróneos que no se corresponderán con los nuevos valores ordinales del tipo enumerado. Para evitar este problema potencial, podemos forzar a la base de datos a utilizar una columna de texto en lugar de una columna numérica: de esta manera, el valor almacenado será el nombre del valor enum, y no su valor ordinal:

```
@Enumerated(EnumType.STRING)
private Genero genero;
```

1.9. Transient

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 metodo getter:

```
public int getEdad() {
   // calcular la edad y devolverla
```

```
}
```

1.10. Colecciones básicas

Una entidad puede tener propiedades de tipo `java.util.Collection` y/o `java.util.Map` que contengan tipos básicos (todos los tipos wrapper, `String`, `BigDecimal`, `BigInteger`, tipos enumerados y tipos insertables).

Los elementos de estas colecciones serán almacenados en una tabla diferente a la que contiene la entidad donde están declarados.

El tipo de colección tiene que ser concreto (como `ArrayList` o `HashMap`) y nunca una interface.

```
private ArrayList<String> comentarios;
```

El código anterior es mapeado por defecto con unos valores predeterminados por JPA. Y por supuesto, podemos cambiar dicha configuración por defecto mediante diversas anotaciones, entre las que se encuentran:

```
@ElementCollection(fetch = FetchType.LAZY)
@CollectionTable(name = "TABLA_COMENTARIOS")
private ArrayList<String> comentarios;
```

`@ElementCollection` permite definir el tipo de lectura (temprana o demorada), así como la clase de objetos que contiene la colección (obligatorio en caso de que la colección no sea de tipo genérico).
Por otro lado, `@CollectionTable` nos permite definir el nombre de la tabla donde queremos almacenar los elementos de la colección. Si nuestra colección es de tipo `Map`, podemos añadir la anotación `@MapKeyColumn(name = "NOMBRE_COLUMNA")`, la cual nos permite definir el nombre de la columna donde se almacenarán las claves del `Map`.

1.11. Tipos insertables

Los *tipos insertables* (embeddable types) son objetos que no tienen identidad, por lo que para ser persistidos deben ser primero insertados dentro de una entidad (u otro insertable que será a su vez insertado en una entidad, etc). Cada propiedad del tipo insertable será mapeada a la tabla de la entidad que lo contenga, como si fuera una propiedad declarada en la propia entidad.
Definimos un tipo insertable con la anotación `@Embeddable`:

```
@Embeddable
public class Direccion {
   private String calle;
```

```
    private int codigoPostal;

    // ...
}
```

Y lo *insertamos* en una entidad (u otro tipo insertable) con la anotación `@Embedded`

```
@Entity
public class Persona {
    // ...

    @Embedded
    private Direccion direccion;
}
```

Ahora, la tabla que contenga las entidades de tipo `Persona` contendrá sus propias columnas, más las definidas en el tipo insertable `Direccion`.

1.12. Tipos de acceso

JPA permite definir dos tipos de acceso:

- Acceso a variable (Field access)
- Acceso a propiedad (Property access)

El tipo de acceso que usará una entidad está definido por el lugar donde situemos sus anotaciones de mapeo.

Si las anotaciones están en las variables que conforman la clase (como hemos hecho hasta ahora), estaremos indicando a JPA que debe realizar *acceso a variable*.

Si, por el contrario situamos las anotaciones de mapeo en los métodos getter, estaremos indicando un *acceso a propiedad*. A efectos prácticos, no existe diferencia alguna entre ambas opciones (más allá de gustos personales y de organización de código). Sin embargo, en determinadas ocasiones debemos ser consecuentes con el tipo de acceso que elijamos, evitando mezclar tipos de acceso (lo cual puede inducir a JPA a actuar de forma errónea).

Un ejemplo típico es el uso de clases insertables: salvo que se especifique lo contrario, las clases insertables heredan el tipo de acceso de la entidad donde son insertadas, ignorando cualquier anotación que se haga hecho sobre ellas previamente. Veamos un ejemplo donde se muestra este problema:

```
@Embeddable
public class Insertable {
    private int variable;

    @Column(name = "VALOR_DOBLE")
    public int getVariable() {
        return variable * 2;
    }

    public void setVariable(int variable) {
```

```
      this.variable = variable;
   }
}

@Entity
public class Entidad
   @Id
   @GeneratedValue
   private Long id;
   @Embedded
   private Insertable insertable;
}
```

En el ejemplo anterior, la clase `Insertable` define un tipo de acceso a propiedad (ya que las anotaciones se han definido en los métodos getter), y queremos que se acceda a través de estos métodos al valor de las variables (tal vez necesitemos realizar cierto procesamiento sobre la variable, como multiplicarla por dos antes de devolverla). Sin embargo, la clase `Entidad` define un tipo de acceso a variable (ya que las anotaciones se han definido en las variables). Puesto que el tipo insertable va a heredar el tipo de acceso de la entidad donde se encuentra definido, cuando accedamos al valor de `Entidad.insertable.variable` obtendremos un valor no esperado (el valor de `variable`, en lugar de su valor multiplicado por dos que devuelve `getVariable()`). Para evitar estos problemas debemos indicar explícitamente el tipo de acceso de `Insertable` mediante la anotación `@Access`:

```
@Embeddable
@Access(AccessType.PROPERTY)
public class Insertable { ... }
```

Anotando la clase `Insertable` con `@Access(AccessType.PROPERTY)`, cuando accedamos a `Entidad.insertable.variable` lo haremos a través de su método getter, a pesar de que en `Entidad` se está usando un tipo de acceso a variable. Un efecto de definir explícitamente el tipo de acceso, es que las anotaciones que no correspondan con ese tipo de acceso serán ignoradas (a no ser que vengan acompañadas a su vez con la anotación `@Access`):

```
@Embeddable
@Access(AccessType.FIELD)
public class Insertable {
   private int variable;

   @Column(name = "VALOR_DOBLE")
   public int getVariable() {
      return variable * 2;
   }

   public void setVariable(int variable) {
      this.variable = variable;
   }
}
```

En el ejemplo anterior, se ha definido un acceso a variable de forma explícito (con `@Access(AccessType.FIELD)`), por lo que la anotación `@Column` será ignorada (no accederemos a la variable a través del método getter, a pesar de estar anotado). Toda la discusión sobre tipos de acceso está directamente relacionada con la capacidad de JPA para acceder a todas las variables y métodos de una clase, independientemente de su nivel de acceso (`public`, `protected`, `package-default`, o `private`).

2. Asociaciones y herencia

2.1. Asociaciones

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

- Unidireccionales
- 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* representan 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 pueden haber en cada extremo de la asociación.

2.2. Asociaciones unidireccionales

Para entender el concepto de unidireccionalidad nada mejor que mostrar ejemplo de asociación unidireccional:

```
@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
}
```

Como puedes ver en el ejemplo anterior:
- cada entidad `Cliente` mantiene una referencia a una entidad `Direccion`. Esta relación es de tipo *uno-a-uno* (one-to-one) unidireccional (puesto que *una* entidad `Cliente` contiene *una* referencia a una entidad `Direccion`, 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 `Direccion`). Esta columna es una *clave foránea* (foreign key), un concepto muy importante en bases de datos relacionales.

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 foráneas 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 customizar este proceso definiendo el nombre de la columna que contendrá la clave foránea mediante la anotación `@JoinColumn`:

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

Otro tipo de asociación muy común es la de tipo *uno-a-muchos* (one-to-many) unidireccional:

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany
    private List<Direccion> direcciones;

    // Getters y setters
}
```

En el ejemplo anterior, cada entidad de tipo `Cliente` mantiene una lista de direcciones a través de la propiedad `direcciones`. Puesto que un objeto `List` puede albergar múltiples objetos en su interior, debemos anotarlo con `@OneToMany`. En este caso, en vez de utilizar una clave foránea en `Cliente`, JPA utilizará por defecto una *tabla de unión* (join table). Cuando ocurre esto, las tablas donde se almacenan ambas entidades contienen una clave foránea a una tercera tabla con dos columnas; esta tercera tabla es llamada *tabla de unión*, y es donde se constata la asociación entre las entidades relacionadas.

```
@OneToMany
@JoinTable(name = ...,
      joinColumn = @JoinColumn(name = ...),
      inverseJoinColumn = @JoinColumn(name = ...))
private List direcciones;
```

2.3. 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` sobre la anotación de asociación dentro de la entidad 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 `@ManyToMany`. Únicamente el cuarto tipo de relación, `@ManyToOne`, no permite el uso de este atributo.

2.4. Lectura temprana y lectura demorada de asociaciones

Las asociaciones son parte del mapeo relacional, y por tanto también son afectadas por este concepto.
El tipo de lectura por defecto para las relaciones *uno-a-uno* y *muchos-a-uno* es temprana (*eager*).
Por contra, el tipo de lectura para los dos tipos de relaciones restantes (*uno-a-muchos* y *muchos-a-muchos*), es demorada (*lazy*).

Por supuesto, ambos comportamientos pueden ser modificados:

```
@OneToMany(fetch = FetchType.EAGER)
private List<Pedido> pedidos;
```

En el caso de las asociaciones, donde se pueden ver involucrados muchos objetos, leerlos de manera temprana puede ser, además de innecesario, inadecuado. En otros casos una lectura temprana puede ser necesaria: si la entidad que es dueña de la relación se desconecta de gestor de persistencia sin haber inicializado aún una asociación con lectura demorada, al intentar acceder a dicha asociación se producirá una excepción de tipo `LazyInitializationException` (podemos encontrar una analogía a este comportamiento cuando tenemos un objeto sin inicializar (con valor `null`), que al no apuntar a ninguna instancia en memoria produce un error si es usado). Sea como sea, configura siempre los tipos de lectura de la forma más adecuada para tu aplicación.

2.5. Ordenación de asociaciones

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

```
@OneToMany
@OrderBy("nombrePropiedad asc")
private List<Pedido> 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).

2.6. Herencia

En las aplicaciones Java, el concepto de herencia es usado de forma intensiva. Por supuesto, JPA nos permite gestionar la forma en que nuestros objetos son mapeados cuando en ellos interviene el concepto de herencia. Esto puede hacerse de maneras distintas:

- Una tabla por familia (comportamiento por defecto)
- Unión de subclases
- Una tabla por clase concreta

➢ El mapeo por defecto es *una tabla por familia*. Una familia no es, ni más ni menos, que todas las subclases que están relacionadas por herencia con una clase madre, inclusive. Todas las clases que forman parte de una misma familia son almacenadas en una única tabla. En esta tabla existe una columna por cada atributo de cada clase y subclase de la familia, además de una columna adicional donde se almacena el tipo de clase al que hace referencia cada fila. Imaginemos el ejemplo siguiente:

```
@Entity
public class SuperClase {
   @Id
   @GeneratedValue
   private Long id;
   private int propiedadUno;
   private String propiedadDos;

   // Getters y Setters
}

@Entity
public class SubClase extends SuperClase {
   @Id
   @GeneratedValue
   private Long id;
   private float propiedadTres;
   private float propiedadCuatro;

   // Getters y setters
}
```

En el ejemplo anterior, tanto las instancias de las entidades `SuperClase` y `SubClase` seran almancenadas en una única tabla que tendrá el nombre por defecto de la clase raíz (`SuperClase`). Dentro de esta tabla habrá seis columnas que se corresponderán con:

- Una columna para la propiedad `id` (válida para ambas entidades, pues en ambas se mapearía a una columna con el mismo nombre)
- Cuatro columnas para las propiedades `propiedadUno`, `propiedadDos`, `propiedadTres` y `propiedadCuatro`
- Una última columna discriminatoria, donde se almacenará el tipo de clase al que hace referencia cada fila

La columna discriminatoria suele contener el nombre de la clase. Esta columna es necesaria para que al recuperar un objeto desde la base de datos, JPA sepa que clase concreta debe instanciar, y que propiedades leer. Las propiedades que son propias de cada clase no deben

ser configuradas como *not null*, ya que al intentar persistir un objeto de la misma familia pero de otra clase (y que tal vez no dispone de las mismas propiedades) obtendríamos un error. Aunque una tabla por familia es el comportamiento por defecto cuando mapeamos entidades en las que interviene la herencia entre clases, podemos declararlo explícitamente mediante el atributo `SINGLE_TABLE` (tabla única) de la anotación `@Inheritance`:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class SuperClass { ... }
```

Ten presente que los valores definidos en `@Inheritance` son heredados por todas las subclases de la entidad anotada, aunque estas pueden sobreescribir dichos valores si desean adoptar una política de mapeo diferente. En lo que refiere al nombre de la columna discriminatoria y a su tipo, estos son por defecto *DTYPE* y *String* (este último es un tipo SQL, evidentemente). Podemos cambiar ambos mediante las anotaciones `@DiscriminatorColumn` y `@DiscriminatorValue`:

```
@Entity
@Inheritance
@DiscriminatorColumn(name = "...", discriminatorType = CHAR)
@DiscriminatorValue("C")
public class SuperClase { ... }
```

La anotación `@DiscriminatorColumn` solo debe ser usada en la clase raiz de la herencia (a no ser que una subclase desee cambiar los parámetros del mapeo, en cuyo caso ella se convertiría en clase raiz por derecho). El atributo `discriminatorType` de la citada anotación nos permite cambiar el tipo de valor que almacenará la columna discriminatoria. Los tipos soportados son `STRING` (por defecto), `CHAR` e `INTEGER`. Si cambiamos en la clase raíz el tipo de valor que almacenará la columna discriminatoria a otro distinto de `STRING`, cada subclase tendrá que indicar de forma explicita el valor que lo representará en la columna discriminatoria. Esto lo hacemos mediante la anotación `DiscrimitadorValue`:

```
@Entity
@DiscriminatorValue("S")
public class SubClase extends SuperClase { ... }
```

En el ejemplo anterior, la columna discriminatoria (que es de tipo CHAR) contendrá un valor *C* si la instancia correspondiente es de tipo `SuperClase`, y una *S* si es de tipo `SubClase`. Por supuesto, no estaría permitido usar `@DiscriminatorValue("C")` si se ha definido una columna discriminatoria de tipo `INTEGER`, etc.

- El segundo tipo de mapeo cuando existe herencia es *unión de subclases*, en el que cada clase y subclase (sea abstracta o concreta) será almacenada en su propia tabla:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class SuperClase { ... }
```

La tabla raíz contiene una columna con una clave primaria usada por todas las tablas, así como la columna discriminatoria. Cada subclase almacenará en su propia tabla únicamente sus atributos propios (nunca los heredados), así como una clave foránea que hace referencia a la clave primaria de la tabla raíz. La mayor ventaja de este sistema es que es intuitivo. Por contra, su mayor inconveniente es que, para construir un objeto de una subclase, hay que hacer una (o varias) operaciones *JOIN* en la base de datos, de manera que se puedan unir los atributos de la subclase con los de sus superclases. Por tanto, una subclase que esté varios niveles por debajo de la superclase en la herencia, necesitará realizar múltiples operaciones *JOIN* (una por nivel), lo cual puede producir un impacto en el rendimiento que tal vez no deseemos.

- El tercer y último tipo de mapeo cuando existe herencia es *una tabla por clase concreta*. Mediante este comportamiento, cada entidad será mapeada a su propia tabla (incluyendo todos los atributos propios y heredados). Con este sistema no hay tablas compartidas, columnas compartidas, ni columna discriminatoria:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class SuperClase { ... }
```

El único requerimiento al utilizar una tabla por clase concreta es que todas las tablas dentro de una misma familia compartan el valor de la clave primaria. De esta manera, cuando almacenemos una subclase, tanto su tabla como las de sus superclases contendrán los mismos valores para las propiedades comunes, gracias a que todas ellas comparten la misma ID. Este sistema puede provocar en determinadas circunstancias problemas de rendimiento, ya que al igual que con la unión de subclases, la base de datos tendrá que realizar múltiples operaciones *JOIN* ante determinadas solicitudes.

2.7. Mapped Superclasses, clases abstractas y no-entidades

Veamos de forma muy superficial la forma en la que gestiona JPA tres tipos de clases no vistas hasta ahora: Mapped Superclases, clases abstractas, y no-entidades.
Mapped Superclasses son clases que no son manejadas por el proveedor de persistencia, pero comparten sus propiedades con cualquier entidad que extienda de ellas:

```
@MappedSuperclass
@Inheritance(strategy = InheritanceType.XXX)
public class MapedSuperClase {
   private String propiedadUno;
   private int propiedadDos;

   // Getters y setters
}

@Entity
public class SuperClase extends MappedSuperClase { ... }
```

En el ejemplo anterior, únicamente `SuperClase` será manejada por el proveedor de persistencia. Sin embargo, durante el mapeo se incluirán todas las propiedades heredadas de `MappedSuperClase`.
Por otro lado, todas las *clases abstractas* son tratadas exactamente igual que si fueran clases concretas, y por tanto son entidades 100% usables siempre que sean declaradas como tal (mediante `@Entity`). Precisamente por esto último, toda clase que no sea declarada como entidad, será ignorada a la hora de realizar el mapeo relacional. A este tipo de clases se las conoce como *no-entidades*.

3. Persistencia y transacciones

Vamos a ver cómo realizar la persistencia, lo que nos permitirá entender mejor qué son los métodos *callback* y las clases *listener*. También introduciremos el concepto de transacción.

3.1. Persistencia

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 alla 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`.

Comencemos definiendo una entidad que manejaremos a través de los futuros ejemplos:

```
@Entity
public class Pelicula {
   @Id
   @GeneratedValue
   private Long id;
   private String titulo;
   private int duracion;

   // Getters y Setters
}
```

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.

En nuestro caso, este archivo mostraría este aspecto:

```
<!--?xml version="1.0" encoding="UTF-8"?-->
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="introduccionJPA" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>Pelicula</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY">
      <property name="eclipselink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver">
      <property name="eclipselink.jdbc.url" value="jdbc:derby://localhost:1527/introduccionJPA_DB;create=true">
      <property name="eclipselink.jdbc.user" value="APP">
      <property name="eclipselink.jdbc.password" value="APP">
      <property name="eclipselink.ddl-generation" value="create-tables">
    </property></property></property></property></property></property></properties>
  </persistence-unit>
</persistence>
```

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.

3.2. Transacciones

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

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

En ella se indica el nombre de la unidad de persistencia (el 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 aplicacion (RESOURCE_LOCAL)
- Manejada por el contenedor (JTA)

Nosotros hemos decidido manejar las transacciones desde la aplicación, y por tanto lo haremos desde nuestro código a través de la interface `EntityTransaction`.

3.3. 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 .

3.4. Persistir una entidad

Ya estamos listos para persistir nuestra primera entidad:

```
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 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 `Pelicula`.
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 metodo `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, `pelicula` 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 `pelicula` 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 lleve 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();
// 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 ésta terminase fallando (por ejemplo durante el bloque `// otras operaciones`) 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 `pelicula` en la línea anterior.

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

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

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.

3.5. Leer una entidad

Ahora que ya sabemos como persistir una entidad en nuestra base de datos, demos un paso más y veamos como realizar el proceso inverso: 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
- Obteniendo una referencia a los datos persistidos

a) Mediante el primer mecanismo los datos son leídos (por ejemplo, con el método `find()`) desde la base de datos y almacenados en una instancia de la entidad:

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

b) Por contra, el segundo mecanismo 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, más concretamente en el primer acceso a cada propiedad y no en el momento de la creación de la entidad:

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

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 mas dinámico para buscar entidades persistidas: JPQL.

3.6. Actualizar una entidad

Antes de explicar como actualizar una entidad, vamos a explicar como 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, ésta 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 `pelicula` 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).

3.7. Eliminar una entidad

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 colector 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 eliminamos 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 *huerfanas* (aquellas que siguen estando persistidas pero sin ninguna entidad que las referencie). 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.

3.8. Operaciones en cascada

La operación de eliminación de entidades huérfanas que acabamos de ver es un tipo de operación llamada *en cascada*. Este tipo de operaciones permiten establecer la forma en que deben propagarse ciertos eventos (como persistir, eliminar, actualizar, etc) entre entidades que forman parte de una asociación. La forma general de establecer como se realizarán estas operaciones en cascada se define mediante el atributo `cascade` de las anotaciones de asociación:

```
@OneToOne(cascade = CascadeType.REMOVE)
private Descuento descuento;
```

El ejemplo anterior es similar a su predecesor. En él, el tipo de operación en cascada que deseamos propagar (`CascadeType.REMOVE`) se indica mediante constantes de la clase `CascadeType`. Estas constantes pueden tener los siguientes valores:

- `PERSIST`
- `REMOVE`
- `MERGE`
- `REFRESH`
- `DETACH`
- `ALL`

La última constante, `CascadeType.ALL`, hace referencia a todas las posibles operaciones que pueden ser propagadas, y por tanto engloba en si misma el comportamiento del resto de constantes.

También podemos configurar varias operaciones en cascada de la lista superior usando un array de constantes `CascadeType`:

```
@OneToOne(cascade = {
      CascadeType.MERGE,
      CascadeType.REMOVE,
      })
private Descuento descuento;
```

Recuerda que cualquier entidad que utilices en tu código debe estar debidamente registrada en el archivo *persistence.xml*, de manera que JPA pueda gestionarla:

```
<class>es.clase.jpa.Descuento</class>
```

3.9. 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:

- Eventos de persistencia (métodos callback asociados anotados con @PrePersists 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. Al escribir métodos callback, debemos seguir algunas reglas para que nuestro código sea válido:

- 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 excepciones de tipo *checked*
- Un método callback no puede invocar métodos de las clases `EntityManager` y/o `Query`

Ten presente que cuando existe herencia entre entidades, los métodos *@Pre/PostXxx* de las superclases son invocados antes que los de las subclases. Así mismo, los eventos de ciclo de vida se propagan en cascada cuando existen asociaciones.

4. JPQL

JPQL (Java Persistence Query Language, Lenguaje de Consulta de Persistencia en Java), un potente lenguaje de consulta orientado a objetos que va incluido con JPA. Aunque no es imprescindible, es recomendable tener unos conocimientos mínimos de lenguaje SQL.
El lenguaje de consultas JPQL recuerda en su forma al SQL estándar, pero en lugar de operar sobre tablas de la base de datos lo hace sobre entidades.

4.1. JPQL básico

Como vimos, 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 `Pelicula` 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 `Pelicula`, y ese alias es usado por la primera `p` (llamada *expresion*) 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
```

La expresiones JPQL utilizan la notación de puntos:

```
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 devuelven, o un único valor, o un conjunto de ellos. Podemos eliminar los resultados duplicados mediante la clausula `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.

4.2. Sentencias condicionales

Mediante una consulta condicional, restringimos los resultados devueltos por una consulta, en base a ciertos criterios lógicos :

```
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. Las sentencias condicionales pueden contener 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`.
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')
```

Gracias a la sentencia anterior se obtendrían todas las instancias de `Pelicula` con una duración menor a 120 minutos que *no* (`NOT`) son del género `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 simbolo 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). Referencia completa de JPQL (en inglés).

4.3. 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 posicion (`?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.

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

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

La sentencia anterior podría tener una claúsula `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 clausula `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*).

4.4. 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.

4.5. 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. Ni que decir tiene que las sentencías `UPDATE` y `DELETE` deben ser usadas con cierta precaución, sobre todo cuando trabajamos con información que se encuentra en producción.

4.6. 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 como crear un objeto `Query` y realizar una consulta a la base de datos:

```
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).

4.7. 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.duracion` y `p.genero`, que en la sentencia JPQL original se corresponden con los parámetros por posicion `?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 una cadena de texto donde se espera un valor numérico), la aplicación lanzará una excepción de tipo `IllegalArgumentException`. Esto también sucederá 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) es tan fácil de aplicar como su versión numérica:

```
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 `:duracion` 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, mantener, y un largo etcétera de ventajas.

4.8. Consultas con nombre (estáticas)

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), como puedes ver en este ejemplo:

```
@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 segundo método de la lista que vimos en la sección 4.7: `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.

4.9. Consultas nativas SQL

El tercer y último tipo de consultas que nos queda por ver requiere una sentencia SQL nativa en lugar de una sentencia JPQL:

```
String sql = "SELECT * FROM PELICULA";
Query query = em.createNativeQuery(sql);
// ...
```

Las consultas SQL nativas pueden ser definidas de manera estática como hicimos con las consultas con nombre, obteniendo los mismos beneficios de eficiencia y rendimiento. Para ello, necesitamos utilizar, de nuevo, metadatos:

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

Tanto las consultas con nombre como las consultas nativas SQL estáticas son muy útiles para definir consultas inmutables (como buscar todas las instancias de una entidad en la base de datos). Las candidatas son aquellas consultas que se mantienen inmutables entre distintas ejecuciones del programa (sin parámetros dinámicos), y que son usadas frecuentemente.