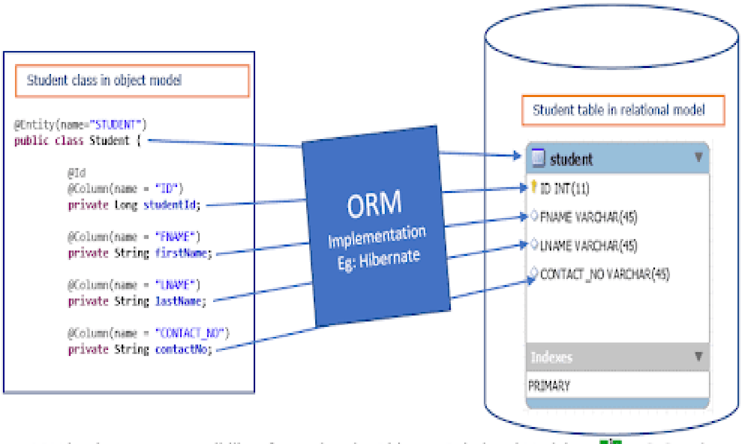
U.T. 3: MAPEO OBJETO-RELACIONAL. HIBERNATE



ORM implements responsibility of mapping the Object to Relational Model. JavaByDeveloper

1. DESFASE OBJETO-RELACIONAL

Una célebre frase de la autora americana Esther Dyson dice: “Usar tablas para almacenar objetos es como conducir tu coche para volver a casa y luego desmontarlo para guardarlo en el garaje. Se puede volver a montar por la mañana, pero finalmente uno se acaba preguntando si esta es la forma más eficiente de aparcar un coche”

El *desfase objeto-relacional* surge cuando en el desarrollo de una aplicación con un lenguaje orientado a objetos se hace uso de una base de datos relacional. Hay que tener en cuenta que esta situación se da porque, tanto los lenguajes orientados a objetos como las bases de datos relacionales, están ampliamente extendidos y por la dificultad de adaptar los datos en el formato de uno y otro.

En cuanto al desfase, ocurre que en nuestra aplicación Java (como ejemplo de lenguaje Orientado Objetos) tendremos, por ejemplo, la definición de una clase cualquiera con sus atributos y métodos:

```java
public class Personaje {
  private int id;
  private String nombre;
  private String descripcion;
  private int vidas;
  private int ataque;

  public Personaje(. . .) {
    . . .
  }

  // getters y setters
}
```

Mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters o setters) cada uno de los atributos con cada uno de los campos (y viceversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente.

```sql
CREATE TABLE personajes (
  id INT PRIMARY KEY AUTO_INCREMENT;
  nombre VARCHAR(50) NOT NULL,
  descripcion VARCHAR(50),
  vidas INT DEFAULT 10,
  ataque INT DEFAULT 10;
);
```

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.

2. TÉCNICAS DE PERSISTENCIA

Para persistir los objetos de nuestra aplicación existen diferentes técnicas con diferentes niveles de abstracción y automatización. Veamos 3 de ellas.

2.1. JDBC: persistencia nativa

Esta técnica consiste en agregar en cada clase instanciable los métodos que se encarguen de guardar y recuperar los atributos del propio objeto en la base de datos. Estos métodos, generalmente denominados *save()*, *read()*, *update()*, *delete()* o nombres similares, ejecutan sentencias SQL que realizan directamente las operaciones necesarias.

El problema de esta técnica es que incumple las normas del buen diseño, ya que incorpora a las clases métodos que no pertenecen a la abstracción que se pretende modelar de la realidad. Al no cumplir con los patrones de diseño también dificultan su mantenimiento y adaptabilidad a cambios.

2.2. DAO: patrón de diseño

En este caso, cada clase permanecerá fiel a la abstracción deseada, sin los métodos necesarios en la técnica anterior. Pero habremos de crear, para cada clase, otra complementaria cuyo único cometido será guardar, actualizar, eliminar y recuperar los datos de un objeto de su clase equivalente.

Aplicado, por ejemplo, a una clase *Empleado*, se crearía otra clase denominada algo así como *EmpleadoDAO* (del patrón DAO, Data Access Object, el cual propone separar por completo la lógica de negocio de la lógica para acceder a los datos). Esta clase contendría, al menos, los métodos:

- Empleado read(String dni), leería y retornaría de la base de datos el empleado, indicado a través de su parámetro DNI.
- void save(Empleado e), guardaría el objeto en la base de datos
- void update(Empleado e), actualizaría el objeto en la base de datos
- void delete(String dni), lo eliminaría de ella.

El principal inconveniente de esta técnica es que casi todo el trabajo recae en el programador, que debe implementarla.

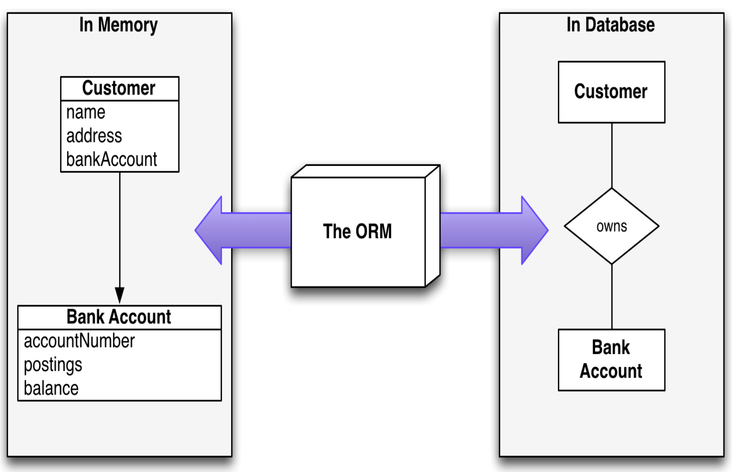
2.3. JPA: Framework de persistencia

Esta es la técnica más frecuentemente utilizada y la que vamos a estudiar. Hace ya años que comenzaron a aparecer diversas alternativas que implementan, de distinta forma pero con similitudes, las mismas funcionalidades. Ante este panorama, fue la propia comunidad de Java la que solicitó el desarrollo de una especificación común; así es como apareció JPA (Java Persistence API), a la cual se deben ajustar las distintas soluciones.

Por tanto, JPA es un estándar que necesita ser implementado por desarrolladores o empresas. Al ser una especificación incluida en Java EE 5 cualquier servidor de aplicaciones compatible con Java EE debe proporcionar una implementación de este estándar. Por otro lado, dado que también es posible utilizar JPA en Java SE, existen bastantes implementaciones de JPA en forma de librerías Java (ficheros JAR) disponibles para incluir en aplicaciones de escritorio o aplicaciones web. La más popular es **Hibernate**, que también se incluye en el servidor de aplicaciones WildFly (anteriormente conocido como JBoss). Otras implementaciones gratuitas son Apache OpenJPA (incluida en el servidor de aplicaciones Jeronimo), Oracle TopLink (incluida en el servidor de aplicaciones GlassFish de Sun) o EclipseLink, de la fundación Eclipse.

Una alternativa a JPA también sería MyBatis, la cual permite un control más directo sobre las operaciones SQL.

3. ¿QUÉ ES EL MAPEO OBJETO-RELACIONAL Y JPA?



Si trabajamos directamente con JDBC tendremos que descomponer el objeto para construir la sentencia `INSERT` del siguiente ejemplo

```
. . .
String sentenciaSql = "INSERT INTO personajes (nombre, descripcion, vidas, ataque)" +
  ") VALUES (?, ?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sentenciaSql);
sentencia.setString(1, personaje.getNombre());
sentencia.setString(2, personaje.getDescripcion());
sentencia.setInt(3, personaje.getVidas());
sentencia.setInt(4, personaje.getAtaque());
sentencia.executeUpdate();

if (sentencia != null)
  sentencia.close();
. . .
```

En cambio, si contamos con un *framework* como *EclipseLink* o *Hibernate*, esta misma operación se traduce en unas pocas líneas de código en las que podemos trabajar directamente con el objeto Java, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que indican con qué tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente. Dicho mapeado lo podemos realizar de alguna de las dos formas siguientes:

- mediante ficheros XML que, para cada clase, especifiquen los parámetros del mapeado
- **mediante anotaciones Java** en el propio código fuente de las clases. Optaremos por estudiar esta última forma.

Veamos un primer ejemplo donde se utilizan anotaciones Java para el mapeado.

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

```
    private int vida;
    @Column(name="ataque")
    private int ataque;
    @Transient
    private String descripcion;

    public Personaje(. . .) {
      . . .
    }

    // getters y setters
}
```

Así, podemos simplemente establecer una sesión con la base de datos y enviarle el objeto, en este caso invocando al método `save` que se encarga de registrarlo en la Base de Datos.

```
. . .
Personaje p = new Personaje(...);
Session sesion = HibernateUtil.getCurrentSession();  // u openSession()
sesion.beginTransaction();
sesion.save(p);
sesion.getTransaction().commit();
sesion.close();
. . .
```

La idea de trabajar con entidades persistentes ha estado presente en la Programación Orientada a Objetos desde sus comienzos. Este enfoque intenta aplicar las ideas de la POO a las bases de datos, de forma que las clases y los objetos de una aplicación puedan ser almacenados, modificados y buscados de forma eficiente en unidades de persistencia. Sin embargo, aunque desde comienzos de los 80 hubo aplicaciones que implementaban bases de datos orientadas a objetos de forma nativa, la idea nunca ha terminado de cuajar. La tecnología dominante en lo referente a bases de datos siempre ha sido, y lo sigue siendo, los sistemas de gestión de bases de datos relacionales (RDBMS). De ahí que la solución propuesta por muchas tecnologías para conseguir entidades persistentes haya sido realizar un mapeado del modelo de objetos al modelo relacional. JPA es una de estas tecnologías, como ya hemos visto.

JPA, como ya se ha dicho, es una especificación, para Java, que detalla cómo manipular datos relacionales en aplicaciones de Java SE y Java EE. JPA especifica tres grandes bloques:

- la API en si misma, en el paquete *javax.persistence*
- el lenguaje de consultas *JPQL* (Java Persistence Query Language)
- y los datos necesarios para el mapeo objeto-relacional. La configuración de estos metadatos se puede realizar mediante anotaciones (con la forma *@anotacion*) o mediante fichero XML. Nosotros, como ya hemos dicho, sólo vamos a ver la primera forma, es decir, haciendo uso de anotaciones.

Una de las características principales de JPA es su simplicidad mediante el uso de estas anotaciones (característica de Java introducida en su versión 5.0) y configuración por defecto, de forma que el desarrollador sólo tiene que especificar aquellas características que necesita que sean distintas de las de por defecto. Por ejemplo, JPA mapea una clase Java con una tabla de la base de datos usando la anotación *@Entity*. Por defecto el nombre de la tabla coincidirá con el nombre de la clase. Ahora bien, podemos modificar ese nombre utilizando anotaciones adicionales. En este caso *@Table(name="nombre-de-tabla")*. En definitiva, la anotación *@Entity* es obligatoria, mientras que *@Table* es opcional.

Al respecto de las distintas y numerosas anotaciones de Hibernate, podemos utilizar su documentación: https://docs.hibernate.org/stable/annotations/reference/en/html_single/.

Una entidad de persistencia es una clase Java ligera, las referidas como POJOs, cuyo estado se puede volcar a una tabla en una base de datos relacional. Las instancias de una de estas entidades se corresponden con filas individuales de la tabla. Las entidades suelen tener relaciones con otras entidades, y estas relaciones se expresan también a través de anotaciones o ficheros XML, como hemos dicho. En definitiva, será nuestra manera de indicar cómo se deben convertir los objetos a tablas y viceversa.

JPQL (abreviatura de “Java Persistence Query Language”), por su parte es un lenguaje de consulta orientado a objetos, independiente de la plataforma, definido como parte de la especificación JPA. Las consultas se realizan a entidades almacenadas en una base de datos relacional. Estas consultas se parecen en su sintaxis a las consultas SQL, pero trabajan con objetos de entidad en lugar de hacerlo directamente con las tablas de la base de datos. Veamos dos ejemplos:

SELECT e FROM Empleado e

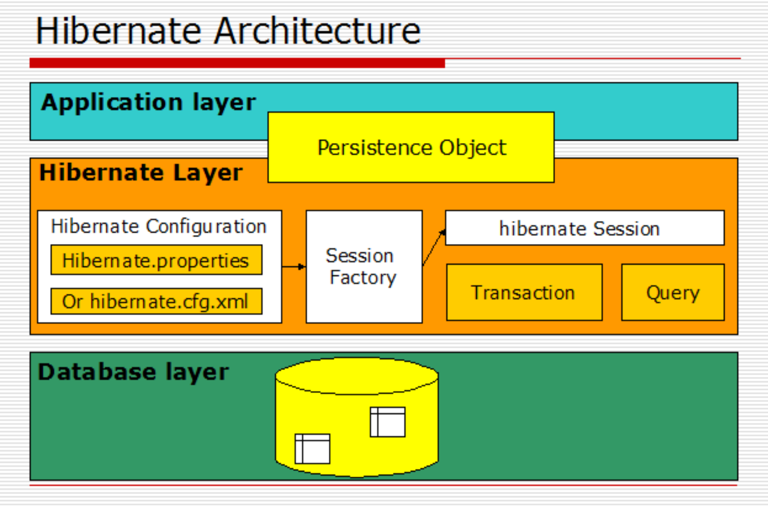
obtendría todos los empleados de la base de datos, mientras que:

SELECT e FROM Empleado e WHERE e.sueldo >= 1000 ORDER BY e.nombre

obtendría los empleados cuyo sueldo iguale o supere los 1000 euros, ordenados por nombre.

En nuestro caso, haciendo uso de Hibernate, utilizaremos su propio lenguaje HQL (Hibernate Query Language).

4. HIBERNATE



En nuestro caso usaremos *Hibernate* como librería *ORM*. Algunas clases/métodos pueden variar entre diferentes versiones de este *framework*, especialmente a la hora de configurar (`hibernate.cfg.xml`) e implementar el gestor de sesiones (`HibernateUtil.java`).

Respecto al fichero de configuración JPA propone el uso del fichero *persistence.xml*, mientras que en Hibernate se utiliza el citado *hibernate.cfg.xml*. Este último es específico de Hibernate para indicar las propiedades de configuración no cubiertas por JPA. Eso incluye la configuración de la conexión a la base de datos, los mapeados de las clases con las tablas y otras opciones avanzadas de Hibernate.

En cuanto a la documentación de Javadoc la tenemos accesible en https://docs.hibernate.org/orm/7.1/javadocs/.

4.1. Configuración

El fichero de configuración de hibernate `hibernate.cfg.xml` se debe crear directamente dentro de la carpeta `src` del proyecto (o en *src/main/resources*, cuando hacemos uso de Maven), y el propio *Hibernate* será el encargado de leerlo para obtener las sesiones que permitan conectar con la base de datos, usando el código que se implementa en el fichero `HibernateUtil.java`. Un ejemplo inicial de `hibernate.cfg.xml` sería:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
```

```
                                                   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
    <property name="hibernate.connection.url">jdbc:postgresql://localhost:5432/basededatos</property>
    <property name="hibernate.connection.username">usuario</property>
    <property name="hibernate.connection.password">contraseña</property>
    <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <!-- Aquí faltarían las líneas para el mapeado de cada clase -->
   </session-factory>
</hibernate-configuration>
```

De las anteriores 7 propiedades, sólo las 4 primeras nos resultarán necesarias en cualquier caso, aunque el parámetro *hibernate.hbm2ddl.auto* también es muy importante. Determina cómo se van a actualizar las tablas de la base de datos cuando Hibernate intente mapearlas con las clases Java. Los posibles valores son los siguientes:

update: se actualiza el esquema de las tablas si ha habido algún cambio en las clases Java. Si no existen, se crean.

validate: se valida que el esquema se puede mapear correctamente con las clases Java. No se cambia nada en el esquema de la base de datos.

create: se crea el esquema, destruyendo los datos previos.

create-drop: se crea el esquema y se elimina al final de la sesión.

Es aconsejable utilizar el valor *update* en desarrollo y el valor *validate* en producción.

La propiedad *hibernate.show_sql* hace que Hibernate muestre por la salida estándar las sentencias SQL que se lanzan al proveedor de base de datos, lo cual nos permitirá confirmar que las operaciones realizadas fueron las previstas.

4.2. Interfaces y clases fundamentales

https://docs.jboss.org/hibernate/orm/6.2/userguide/html_single/Hibernate_User_Guide.html

Para almacenar en objetos los datos desde la base de datos, Hibernate mantiene una sesión (instancia de la clase **Session**) que viene a ser equivalente a la conexión en JDBC. Esta clase incluye métodos como *save*, *createQuery*, *beginTransaction* y *close*. La creación de instancias de *Session* no influye de manera importante en el rendimiento, lo cual nos permitirá crear y destruir sesiones incluso entre distintas peticiones. La sesión nos creará algo así como una caché de objetos cargados en la interacción con la base de datos.

Además de esta clase, tenemos las siguientes interfaces:

- *SessionFactory*, para creación de objetos *Session*. Esta interfaz debe compartirse entre diferentes procesos o hilos de ejecución, y debemos asegurarnos de utilizar una única *SessionFactory* para toda la aplicación. Sólo si nuestra aplicación accediera a varias bases de datos se necesitaría más de una: una para cada.
- *Configuration* se utiliza para aplicar la configuración con las propiedades específicas de Hibernate y el mapeado de objetos.
- *Transaction* nos permitirá operar todos los cambios como una única orden de actualización sobre la base de datos, asegurando que cualquier fallo pueda ser revertido con *rollback*. En nuestro caso, generalmente omitiremos los *try-catch* pertinentes, con el correspondiente *rollback* en el *catch*, por simplificar el código.
- *Query* nos permitirá realizar consultas mediante el lenguaje propio de Hibernate, HQL.

La siguiente clase *HibernateUtil* responde al patrón *Singleton*, asegurándonos que sólo se genera una instancia de *SessionFactory*. Esto debe ser así por eficiencia, se ha de evitar que en cada ejecución se genere una nueva instancia de dicha clase. Para ello se hace el constructor privado y sólo se genera la instancia si la referencia estaba a null.

```java
public class HibernateUtil {

 public static SessionFactory factory;

     // private para deshabilitar la creación de otros objetos desde otras clases
 private HibernateUtil() {
 }
     // nos aseguramos de crear una sola instancia (singleton)
 public static SessionFactory getSessionFactory() {

     if (factory == null) {
         factory = new Configuration().configure().
                 buildSessionFactory(new StandardServiceRegistryBuilder().configure().build());
     }
     return factory;
 }
}
```

La llamada a *Configuration.configure()* carga el fichero de configuración *hibernate.cfg.xml*, inicializando el entorno de Hibernate. Finalmente se creará el objeto Session.

```java
SessionFactory sf = HibernateUtil.getSessionFactory();
Session s = sf.openSession();   // o getCurrentSession()
```

Para obtener un objeto *Session* podemos utilizar *openSession()* o *getCurrentSession()*. Ambos son métodos de *SessionFactory*. Veamos las diferencias entre uno y otro.

Método *openSession()*

- siempre crea un nuevo objeto de sesión
- el programador necesita volcar los cambios en la base de datos y cerrar explícitamente la sesión
- se debe crear un objeto de sesión por solicitud en un entorno multiproceso y también en aplicaciones web. Es decir, un objeto *Session* no puede ser compartido. Por tanto, es el método a utilizar en esos casos
- no es necesario configurar ninguna propiedad para llamar a este método.

Método *getCurrentSession()*

- crea una nueva sesión si no hay ninguna creada en el contexto de hibernación actual
- el programador no necesita volcar los cambios ni cerrar la sesión; Hibernate se encargará de ello internamente
- en un entorno de un solo subproceso, es más rápido que *openSession()*. No soporta trabajo con hilos
- necesita configurar una propiedad adicional en el archivo *hibernate.cfg.xml*, de lo contrario arrojará una excepción

```xml
<session-factory>
   <!--  Put other elements here -->
   <property name="hibernate.current_session_context_class">
      thread
   </property>
</session-factory>
```

La forma de trabajo habitual en Hibernate con Java SE consistirá por tanto en:

- crear u obtener un *Session* a partir del *SessionFactory*
- marcar el comienzo de la transacción

- realizar operaciones sobre las entidades
- hacer el commit sobre la transacción, el cual reflejará todos los cambios en la base de datos
- cerrar la transacción y el objeto *Session*.

Cuando una entidad se obtiene de la base de datos se guarda en una caché en memoria mantenida por Hibernate. Esta caché se denomina **contexto de persistencia**, y se guarda en el objeto *Session* que utiliza la aplicación. De forma similar a las conexiones JDBC, los objetos *Session* son los encargados de gestionar la persistencia de las entidades declaradas en la aplicación.

Todas las entidades que se crean en una sesión son gestionadas por ella y viven en su contexto de persistencia. Cuando la sesión se cierra, las entidades siguen existiendo como objetos Java, pero a partir de ese momento se encuentran desconectadas (*detached*) de la base de datos.

Los cambios en las entidades no se propagan automáticamente a la base de datos, sino al realizar un commit de la transacción. Entonces se chequea el contexto de persistencia, detecta los cambios que se han producido en las entidades, utiliza el proveedor de persistencia para generar las sentencias SQL asociadas a los cambios y vuelca (*flush* o *commit*) esas sentencias en la base de datos.

Pero ¿qué diferencia a **flush()** de *commit()*?. Supongamos el siguiente código:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for (int i = 0; i < 100000; i++) {
    Customer customer = new Customer(...);
    session.save(customer);
    if (i % 20 == 0) { // 20, igual que el tamaño de lote JDBC batch por defecto
        // vuelca un lote de entidades y libera su memoria
        session.flush();
        session.clear();
    }
}
tx.commit();
session.close();
```

Sin la llamada a *flush* previa al *commit*, nos podríamos encontrar con que nuestra caché lanzara una *OutOfMemoryException*. El método *flush()* sincronizará la base de datos con el estado actual de los objetos que se encuentran en el área de persistencia, pero no confirma la transacción. Por lo tanto, si se produce alguna excepción después de llamar a *flush()*, la transacción se revertirá. De esta manera, puede ser más apropiado sincronizar la base de datos con pequeños fragmentos de datos usando *flush()* en lugar de enviar una gran cantidad de datos todos de una usando *commit()* y correr el riesgo de obtener una *OutOfMemoryException*.

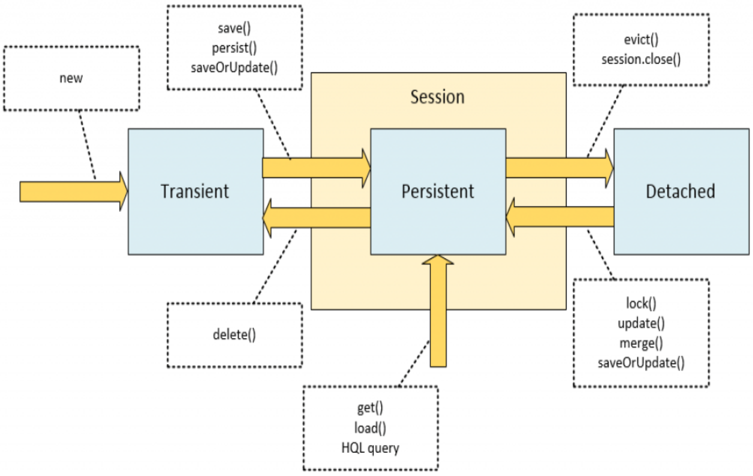
Dicho de otra forma, limpiar la sesión (*flush*) simplemente hace que los datos que se encuentran actualmente en la sesión se sincronicen con los que se encuentran en la base de datos. Sin embargo, el hecho de haber limpiado la sesión no significa que los datos no se puedan revertir.

4.3. Estados de un objeto en Hibernate

Cada objeto gestionado por Hibernate podrá estar en alguno de los 3 estados siguientes:

- Transitorio
- Persistente
- Desligado (*detached*)

Un objeto estará en estado transitorio (transient) cuando se haya instanciado y no se haya asociado todavía a una sesión. No existe por lo tanto todavía en la base de datos. Lo podremos hacer persistente mediante el método *save* o *persist*, o cuando haya sido cargado con los métodos *load* o *get*. Hibernate detectará los cambios sobre estos objetos persistentes y los reflejará en la base de datos con cada *commit*. Por último, los objetos desligados (detached) quedan en este estado cuando cerramos la sesión con *close*. Estas instancias podrán ser asociadas a sesiones futuras.



Entre los métodos *save()* y *persist()*, usados ambos para persistir los objetos en la base de datos, hay algunas diferencias:

- *save()* es propio de Hibernate, mientras que *persist()* lo es de JPA. El segundo, por ello, es más portable.
- el primero inserta inmediatamente en la base de datos, mientras que el segundo no lo hace mientras no hagamos un *commit()* o *flush*.
- el primero puede funcionar fuera de una transacción (aunque no es recomendado).
- el primero asigna el valor de *id* inmediatamente al objeto, mientras que el segundo lo hace después del *commit()*.
- en caso de querer insertar un valor de *id* ya existente, el primero puede modificar el valor existente mientras que el segundo lanzaría una excepción.

4.4. Mapeo de entidades/relaciones con clases/atributos Java

En el caso de las entidades, se deben anotar tanto la propia clase como cada uno de los atributos (con cada anotación sobre el atributo o, alternativamente, sobre su *getter/setter*). Con ello indicamos con qué tabla mapearla y cómo mapear cada atributo con la columna de la tabla que corresponda, respectivamente.

Las anotaciones para clases Java a mapear con una tabla son:

- `@Entity`. Indica que la clase es una tabla en la base de datos
- `@Table(name = "nombre_tabla", catalog = "nombre_base_datos")` Indica el nombre de la tabla y la base de datos a la que pertenece (este último parámetro no es necesario ya que esa información viene en el fichero de configuración)

Ambas anotaciones (*Entity* y *Table*) pueden ser incluidas desde JPA (@javax.persistence.Entity) o desde Hibernate (@org.hibernate.annotations.Entity). Esta última extiende la funcionalidad de la primera.

En el caso de los atributos simples que deben ser mapeados con los campos de la tabla correspondiente:

- `@Id` Indica que un atributo es la clave
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`. La mejor técnica para generar un identificador consiste en delegar esta acción en la base de datos, en función de sus características, en lugar de implementarla nosotros mismos. Esto lo conseguimos con la anotación *@GeneratedValue*, eligiendo alguna de las cuatro estrategias disponibles. IDENTITY confía en el tipo de dato identidad de la base de datos asignando un valor autoincrementable.

- `@Column(name = "nombre_columna")` Se utiliza para indicar el nombre de la columna en la tabla. El campo name en las columnas no es obligatorio si el nombre del campo en la tabla de la base de datos se llama igual.
- `@Transient` Indica que el atributo no será persistido: Hibernate lo ignorará a la hora de guardar cada objeto.
- `@OneToMany`: Sirve para definir una relación uno-a-muchos, en el siguiente ejemplo entre Autor y Mensaje. La propiedad *cascade*, con distintos valores posibles, en este ejemplo se utiliza para indicar que las acciones de borrado, persistencia y mezcla se propagan en cascada a los mensajes. Más adelante explicaremos la anotación *@mappedBy*, así como el resto de anotaciones para otras multiplicidades en las relaciones de asociación.

Hibernate obliga también a definir un constructor vacío en todas las entidades. Si no lo hacemos muestra un mensaje de error. Veamos un ejemplo de una de estas clases.

```
@Entity
@Table(name = "autores")
public class Autor implements Serializable {
        @Id
        @GeneratedValue
        @Column(name = "autor_id")
        Long id;
        private String correo;
        private String nombre;
        @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL)
        private Set<Mensaje> mensajes = new HashSet<Mensaje>();
        @Transient
        private String comentario;

        public Autor() {
        }

        /* otros constructores
        ... */

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        /* más setters y getters
                ... */

        public Set<Mensaje> getMensajes() {
                return mensajes;
        }

        public void setMensajes(Set<Mensaje> mensajes) {
                this.mensajes = mensajes;
        }

        @Override
        public String toString() {
                return "Autor [id=" + id + ", correo=" + correo + ", nombre="
                        + nombre + "]";
```

```
        }
}
```

En el ejemplo estamos definiendo una relación uno a muchos entre autor y mensajes. Estas relaciones se definen en JPA, o Hibernate, definiendo campos del tipo de la otra entidad y anotándolos según el tipo de relación (uno-a-uno, uno-a-muchos o muchos-a-muchos). En nuestro ejemplo relacionamos el autor con todos sus mensajes, igual como en la otra clase *Mensaje*, no mostrada aquí, podríamos relacionar cada mensaje con el autor que lo ha escrito. La definición de estas relaciones facilita mucho la programación porque evita la realización explícita de muchas consultas SQL. Por ejemplo, si en una consulta recuperamos una colección de mensajes, JPA o Hibernate podrá recuperar al mismo tiempo el autor asociado en cada caso y guardarlo en el campo autor. De esta forma podemos utilizarlo inmediatamente sin haber de realizar ninguna consulta adicional.

La cardinalidad de la relación la definimos con la anotación *OneToMany* en el campo mensajes de *Autor* (un autor tiene una colección de mensajes) y su relación inversa *ManyToOne* en el campo autor de *Mensaje* (muchos mensajes pueden tener el mismo autor). Estas anotaciones sirven para realizar el mapeo de la relación a las tablas. En este caso se crea una clave ajena en la tabla de mensajes que apunta al autor de cada mensaje. Esto lo indicamos con la anotación *mappedBy* en el campo mensajes de la clase *Autor*. Si nos fijamos en el esquema SQL de la tabla de autores, si la podemos consultar, veremos que no hay ninguna columna mensajes en ella. La colección con los mensajes de un autor la construye JPA con una consulta SQL sobre la tabla de mensajes y utilizando la clave ajena definida por la anotación *mappedBy* (en este caso el campo autor).

La anotación *ManyToOne*, la correspondiente inversa, se colocaría en el campo *autor* de la clase *Mensaje*, que hace de clave ajena a la tabla de autores. La clase *Mensaje* es la que conocemos como propietaria de la relación.

Las tablas podrán ser creadas con anterioridad a nuestro programa o, alternativamente, con las clases y sus anotaciones podríamos, con la ejecución del programa, crear las tablas. Si lo hacemos de la primera forma tendremos la ventaja de que se puede crear de forma semiautomática las clases POJOs que representen la lógica equivalente dentro de nuestra aplicación desde cualquier IDE; en algunos IDEs veremos una opción similar a crear los “ficheros y mapeado y POJOs a partir de la base de datos”. No obstante, no es tan obvio que hacerlo de esta forma sea mejor opción ya que a veces, corregir los errores que produce, muchos de ellos básicos, no está claro que compense en tiempo.

Por último, que no se nos olvide añadir los “mapping” del fichero de configuración básica de Hibernate, “hibernate.cfg.xml”, para hacer referencia a estas clases:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                                         "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
    <property name="hibernate.connection.url">jdbc:postgresql://localhost/basededatos</property>
    <property name="hibernate.connection.username">usuario</property>
    <property name="hibernate.connection.password">contraseña</property>
    <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQL95Dialect</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class = "com.dam2.ejhibern.Actor"/>
    <mapping class = "com.dam2.ejhibern.Pelicula"/>
   </session-factory>
</hibernate-configuration>
```

Respecto a la propiedad *Cascade* de las anotaciones de cardinalidad, tenemos varios tipos, entre ellos PERSIST, MERGE, DETACH, REMOVE y ALL. Para las entidades nuevas, siempre debemos usar PERSIST, mientras que para las entidades desligadas debemos usar MERGE. Básicamente, PERSIST debe usarse con entidades totalmente nuevas para agregarlas a la base de datos (si la entidad ya existe en la base de datos, se lanzará una excepción EntityExistsException). MERGE será útil cuando volvamos a poner la entidad en el contexto de persistencia si se desligó previamente. DETACH la utilizamos para que al desligar un objeto sus objetos relacionados también se desliguen.

Para las entidades ya administradas, ya guardadas en la sesión, no necesitamos ningún método de guardado porque Hibernate sincroniza automáticamente el estado de la entidad con el registro de la base de datos subyacente.

Mapeado de fechas y horas

Muchas veces cometemos el error de pensar que al guardar una fecha guardamos sólo año, mes y día. Sin embargo, se guardan también la hora, minutos y segundos. ¿Qué hemos de hacer si sólo queremos guardar la fecha o sólo la hora del día, con sus minutos y segundos? Para ello, contamos con los tipos **date** (sólo fecha: día, mes y año), **time** (sólo horas, minutos y segundos) o **timestamp** (todos ellos). Cuando no especificamos ninguno de ellos se aplica por defecto el último de ellos, con lo que se guardará toda la información (fecha y hora del día).

4.5. Operaciones sobre la Base de Datos

Registrar un objeto

Para guardar datos, deberemos crear una sesión (conexión con la Base de Datos), utilizando la clase `HibernateUtil`, y dentro de ella una transacción. A continuación prepararemos un objeto del tipo que nos interese y lo guardaremos con “save” o "persist". Finalmente, deberemos hacer un “commit” de la transacción y cerrar la sesión.

Para registrar ese nuevo objeto en la Base de Datos necesitamos haber creado previamente la clase y haberla mapeado correctamente con la tabla que le corresponda.

```
. . .
UnaClase unObjeto = new UnaClase();
. . .
Session sesion = HibernateUtil.getCurrentSession();
sesion.beginTransaction();
sesion.save(unObjeto);
sesion.getTransaction().commit();
sesion.close();
. . .
```

Hay que tener en cuenta que, entre el inicio y cierre de la transacción, podemos realizar más de una operación y éstas se ejecutarán como tal. Es la forma correcta cuando queramos registrar más de un objeto y éstos estén relacionados de alguna forma en la que dependan entre ellos. Un caso muy claro sería el del registro de un pedido junto con todas sus líneas de detalle puesto que no tendría sentido registrarlo sin los detalles, por lo que la forma más segura sería darlos de alta dentro de una misma transacción.

```
. . .
Session sesion = HibernateUtil.getCurrentSession();
sesion.beginTransaction();
sesion.save(unPedido);
for (DetallePedido detallePedido : detallesDelPedido)
  sesion.save(detallePedido);
sesion.getTransaction().commit();
sesion.close();
. . .
```

Modificar un objeto

En el caso de que queramos modificar un objeto, la operación se realiza de la misma forma que para el caso de registrar uno nuevo. Hibernate decide qué hacer (si registrar o modificar) comprobando si el objeto que se le envía tiene un valor válido para el campo `id`. Así, la única diferencia con el ejemplo anterior es que ahora dispondremos de un objeto que hemos obtenido previamente de la base de datos y al que hemos realizado algunas modificaciones (nunca en el campo `id`).

```
. . .
Session sesion = HibernateUtil.getCurrentSession();
sesion.beginTransaction();
... // modificar el objeto, por ejemplo con setters
sesion.update(unObjeto);
sesion.getTransaction().commit();
sesion.close();
. . .
```

Recomendaciones:

Para objetos nuevos: usa *persist()*.

Para objetos *managed*, es decir, ya persistidos o existentes en el área de persistencia: no necesitas hacer nada, modifícalo directamente y los cambios se actualizarán en la base de datos en el siguiente *commit*.

Para objetos detached: usa *merge()* (JPA) o *update()* (Hibernate).

No utilices *persist()* para actualizar: siempre lanzará una excepción

El mecanismo de persistencia automática de Hibernate (cambios en objetos managed) es lo más común y recomendado para actualizaciones.

Eliminar un objeto

```
. . .
Session sesion = HibernateUtil.getCurrentSession();
sesion.beginTransaction();
sesion.remove(unObjeto);
sesion.getTransaction().commit();
sesion.close();
. . .
```

5. RELACIONES

El tipo de relación entre clases más habitual es el de **asociación**. Estas se van a definir incluyendo atributos del tipo de la otra entidad y anotándolos según el tipo de relación (uno-a-uno, uno-a-muchos o muchos-a-muchos). Por ejemplo, relacionamos un mensaje con el autor que lo ha escrito y el autor con todos sus mensajes. La definición de estas relaciones facilita mucho la programación porque evita la realización explícita de muchas consultas SQL. Por ejemplo, si en una consulta recuperamos una colección de mensajes, JPA podrá recuperar al mismo tiempo el autor asociado y guardarlo en el campo autor. De esta forma podremos utilizarlo inmediatamente sin tener que realizar ninguna consulta adicional.

Atendiendo a la **direccionalidad**, la relación entre dos entidades puede ser:

- unidireccional: el mapeado se realiza únicamente en una dirección. Uno de los extremos de la relación no sabrá nada acerca del otro extremo.
- bidireccional: ambos extremos de la relación saben del otro. Esta es la opción más usual, puesto que permite navegar por el gráfico de objetos en ambas direcciones.

El extremo **propietario de la relación** será el que mantenga la clave ajena (*foreign key*) de la base de datos. En una relación "Uno a Uno", es el extremo donde queramos especificar la clave ajena. En una relación "Uno a Muchos", o "Muchos a Uno", es el lado del "Muchos".

Lado propietario y lado no propietario: en una relación bidireccional, siempre hay un "lado propietario" y un "lado no propietario". El lado propietario es el que se encarga de actualizar la relación en la base de datos y hará uso de la anotación *@JoinColumn*. Hibernate sólo considera la configuración del lado propietario cuando detecta cambios en la relación.

Para el mapeado de las relaciones, en ambas clases se indicará el tipo de relación visto desde el lado correspondiente:

- `@OneToOne` indica que el objeto es parte de una relación 1-1
- `@ManyToOne` indica que el objeto es parte de una relación N-1. En este caso el atributo sería el lado 1
- `@OneToMany` indica que el objeto es parte de una relación 1-N. En este caso el atributo sería el lado N
- `@ManyToMany` indica que el objeto es parte de una relación N-M. En este caso se indica la tabla que mantiene la referencia entre las tablas y los campos que hacen el papel de claves ajenas en la base de datos

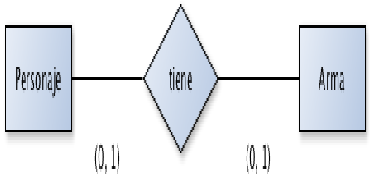
En el otro lado de la relación, el no propietario, indicaremos el tipo de relación acompañado de la anotación `mappedBy` que indique el atributo de la clase propietaria donde se especifica toda la información sobre el mapeo.

El uso de *@JoinColumn* y *mappedBy* permite especificar las columnas que son claves ajenas en una relación. La primera identifica la entidad propietaria de la relación y permite definir el nombre de la columna que define la clave ajena (equivalente a la anotación *@Column* en otros atributos). Su uso es opcional, aunque es recomendable, ya que hace el mapeo más fácil de enteder. La segunda anotación es obligatoria: se usa, como hemos dicho, en el atributo de la entidad no propietaria que no se mapea en ninguna columna de la tabla. Hemos de recordar que consideramos entidad propietaria de la relación a la entidad cuya tabla contiene la clave ajena hacia la otra.

Además de *@JoinColumn* también tenemos las anotaciones *@PrimaryKeyJoinColumn*, y *@MapsId* junto a *@JoinColumn*, a utilizar cuando las dos clases comparten el mismo valor de clave.

A partir de JPA 2.0, *@PrimaryKeyJoinColumn* se utiliza únicamente para compartir claves entre tablas correspondientes a clases con relación de herencia y estrategia *Join* de una tabla por clase (hablamos de ello en el apartado de relaciones de herencia más adelante). Para compartir claves entre tablas sin relaciones de herencia, con relación de asociación uno a uno, utilizaríamos conjuntamente *@MapsId* y *@JoinColumn* en la entidad propietaria de la relación que, además, no ha de incluir la anotación *GeneratedValue*.

5.1. Relaciones 1-1



Veamos el primer ejemplo en el que se hace uso de claves compartidas (observa cómo las correspondientes anotaciones para la relación se pueden indicar no en el atributo, sino en el getter).

```
@Entity
@Table(name = "personajes")
public class Personaje implements Serializable {

     @Id
     @GeneratedValue(strategy = GenerationType.IDENTITY)
     private Long id;
     . . .
     @OneToOne(cascade = CascadeType.ALL, mappedBy = "arma")
     private Arma arma;
     . . .
}
```

```java
@Entity
@Table(name = "armas")
public class Arma implements Serializable {

     @Id
     private Long id;
     . . .
     @OneToOne(cascade = CascadeType.ALL)
     @MapsId
     @JoinColumn
     private Personaje personaje;
     . . .
}
```

Observa cómo en este caso Arma no requiere *@GeneratedValue* para la generación de nuevos ids, ya que han de ser los mismos valores de clave principal que la clase Personaje.

Otro ejemplo, ahora con claves no compartidas:

```java
@Entity
@Table(name="empleados")
public class Empleado implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

. . .
    @OneToOne(mappedBy = "empleado", cascade = CascadeType.PERSIST, orphanRemoval = true)
    private Usuario usuario;
    ...
}
```

```java
@Entity
@Table(name="usuarios")
public class Usuario implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int userId;
. . .
    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name="emplId")
    private Empleado empleado;
    ...
}
```

Una buena práctica es usar cascade en la entidad padre ya que nos permite propagar los cambios y aplicarlos a los hijos. El uso de *cascade* en la entidad padre hace que al persistir un empleado se persista también su usuario. En nuestro ejemplo, Usuario no tiene sentido que exista si Empleado no existe, por lo que Empleado es el que tendrá el rol padre. En la práctica en muchas ocasiones *cascade* irá del lado del *mappedBy* (también para otras cardinalidades en las relaciones).

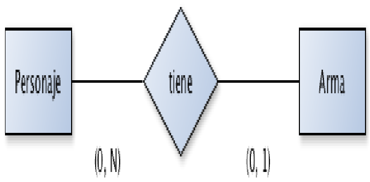
Si observamos el código anterior, hemos decidido que la FK (Foreign Key, clave ajena) la tenga Usuario. Podemos decir que Usuario es el propietario de la relación o propietario de esa FK (owning side) y Empleado, la no propietaria de la relación, no posee esa FK (non-owning side). Pero, ¿Cómo creo una relación bidireccional en caso de que Empleado quiera obtener las propiedades de Usuario? Podríamos pensar en tener otra FK en Empleado apuntando a Usuario pero esto generaría una duplicidad innecesaria en nuestro modelo de base de datos. Para poder realizar este mapeo correctamente están las anotaciones @JoinColumn y mappedBy.

Observa el uso de *orphanRemoval= true* en Empleado. Especifica que la entidad hijo debe ser eliminada automáticamente por el propio ORM si ha dejado de ser referenciada por una entidad padre. Ojo, no confundir con cascadeType que son operaciones a nivel de base de datos.

Usa CascadeType.REMOVE cuando quieres eliminar hijos sólo cuando se elimine el padre. Los hijos pueden ser reasignados a otros padres.

Usa orphanRemoval = true cuando los hijos no tienen sentido sin el padre, la relación es de composición (el hijo es parte integral del padre). Los hijos no se reasignarán nunca a otro padre.

5.2. Relaciones 1-N

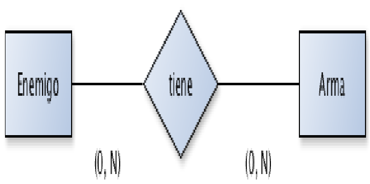


Como ya hemos dicho, en este caso es importante tener en cuenta que el lado propietario de la relación es el de cardinalidad N.

```
@Entity
@Table(name = "personajes")
public class Personaje implements Serializable {
  . . .
  private Arma arma;
  . . .
  @ManyToOne
  @JoinColumn(name="id_arma")
  public Arma getArma() { return arma; }
  . . .
}
```

```
@Entity
@Table(name = "armas")
public class Arma implements Serializable {
  . . .
  private List<Personaje> personajes;
  . . .
  @OneToMany(mappedBy = "arma", cascade = CascadeType.ALL)
  public List<Personaje> getPersonajes() { return personajes; }
  . . .
  }
```

5.3. Relaciones N-M



```java
@Entity
@Table(name = "estudiantes")
public class Estudiante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @ManyToMany
    @JoinTable(
        name = "estudiante_curso", // Nombre de la tabla intermedia
        joinColumns = @JoinColumn(name = "estudiante_id"), // Columna para la clave ajena de Estudiante
        inverseJoinColumns = @JoinColumn(name = "curso_id") // Columna para la clave ajena de Curso
    )
    private Set<Curso> cursos = new HashSet<>();

    // Getters y Setters
}

@Entity
@Table(name = "cursos")
public class Curso {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

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

    // Getters y Setters
}
```

La relación se mapea, a diferencia de los casos anteriores, utilizando una tercera tabla (tabla "join") que construye JPA automáticamente utilizando los nombres de las dos entidades. Es una tabla con dos columnas que contienen claves ajenas a las tablas de las entidades. La primera columna apunta a la tabla propietaria de la relación y la segunda a la otra tabla.

> @ManyToMany: Le estamos diciendo a JPA que la relación entre Estudiante y Curso es de muchos a muchos. En Estudiante, usamos @JoinTable para especificar el nombre de la tabla intermedia (estudiante_curso) y cuáles son las columnas de las claves ajenas. mappedBy: En la entidad Curso, usamos mappedBy = "cursos" para indicar que la relación Many-to-Many está siendo manejada por la otra entidad (Estudiante). Esto significa que Estudiante es el "dueño" de la relación.

¿Y si la tabla intermedia necesita más datos?

A veces, la tabla intermedia no solo conecta dos entidades, sino que también almacena más información. Por ejemplo, podrías querer registrar la fecha en que el estudiante se inscribió en el curso. En este caso, necesitamos una tercera entidad que represente esta tabla intermedia y la relación N a N se transforma en dos relaciones 1 a N.

```java
@Entity
public class Inscripcion {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "estudiante_id")
    private Estudiante estudiante;

    @ManyToOne
```

```
    @JoinColumn(name = "curso_id")
    private Curso curso;

    private LocalDate fechaInscripcion;

    // Getters y Setters
}
```

Con esta configuración, ahora podemos manejar datos adicionales (como la fecha de inscripción) directamente en la entidad Inscripcion y tendríamos que apoyarnos en dos relaciones una a muchos.

5.4. Tipos de captura

Existen distintos tipos de captura o búsqueda (*FetchType*) en función de cómo se efectúa la búsqueda de los datos en la relación:

- Perezosa o demorada (*FetchType.LAZY*): el dato no se solicita hasta que se referencia. Es decir, si se tiene una lista, Hibernate va a esperar a que se haga una consulta sobre la lista para obtener los datos de la base de datos.
- Ansiosa o impaciente (*FetchType.EAGER*): los datos relacionados se obtienen por adelantado.

En general, la estrategia ansiosa reduce el número de accesos a la base de datos al mínimo, ya que se descargan, todos de una, los objetos relacionados, pero es mucho más exigente en consumo de memoria.

Por defecto, de los cuatro tipos de relación entre entidades permitidos en JPA (*@OneToOne*, *@OneToMany*, *@ManyToOne* y *@ManyToMany*) sólo dos son de tipo Lazy: *@OneToMany* y *@ManyToMany*. Esto tiene mucho sentido, pues son estos dos tipos los que a más objetos conectan en el otro lado de la relación. Al ser el comportamiento por defecto, no debemos hacer nada para declarar estas relaciones como Lazy (es su comportamiento implícito). Un ejemplo de uso de uno de estos tipos de captura sería:

```
@Entity
@Table(name = "empleados")
public class Empleado implements Serializable {
    // ...

    @OneToOne(fetch = FetchType.LAZY)
    private Direccion direccion;
}
```

5.5. Relaciones de herencia

En cuanto a las relaciones de herencia entre clases, existen tres posibles estrategias para realizar este mapeo:

- Tabla única
- Tablas join
- Una tabla por clase

La estrategia más común es la de **tabla única**. En ella todas las clases en la jerarquía de herencia se mapean en una única tabla. Esta tabla contiene almacenadas todas las instancias de todos los posibles subtipos. Los distintos objetos en la jerarquía se identifican utilizando una columna especial denominada columna discriminante (*discriminator column*). Esta columna contiene un valor distinto según la clase a la que pertenezca el objeto. Además, las columnas que no se correspondan con atributos de un tipo dado se rellenan con NULL.

Esta estrategia tiene la ventaja de ser la opción que mejor rendimiento da, ya que sólo es necesario acceder a una tabla (aunque está totalmente desnormalizada). Tiene como inconveniente que todos los campos de las clases hijas tienen que admitir nulo.

La segunda estrategia, la de **tablas join** consiste en una tabla para el padre de la jerarquía, con los atributos comunes, y otra tabla para cada clase hija con los atributos propios. Es la opción más normalizada, y, por lo tanto, la más flexible (puede ser interesante si tenemos un modelo de clases muy cambiante), ya que para añadir nuevos tipos basta con añadir nuevas tablas y si queremos añadir nuevos atributos sólo hay que modificar la tabla correspondiente al tipo donde se está añadiendo el atributo. Tiene la desventaja de que para recuperar la información de una clase, hay que ir haciendo join con las tablas de las clases padre.

La tercera consiste en una **tabla por clase**. En este caso cada tabla es independiente, pero los atributos del padre (atributos comunes en los hijos), tienen que estar repetidos en cada tabla. En principio puede tener serios problemas de rendimiento. Sería la opción menos aconsejable.

Para implementar la herencia en JPA se utiliza la anotación *@Inheritance* en la clase padre. En esta anotación hay que indicar la estrategia de mapeado utilizada con el elemento *strategy*. También hay que añadir a la clase padre las anotaciones *@DiscriminatorColumn* y *@DiscriminatorValue*, que indican la columna discriminante. El tipo de la columna discriminante se define con el elemento *DiscriminatorType*, que puede tomar como valor las constantes DiscriminatorType.STRING, DiscriminatorType.INTEGER o DiscriminatorType.CHAR. Esa columna discriminante no se ha de añadir como atributo a la clase, aunque se añadirá automáticamente como columna en la tabla.

El siguiente código muestra cómo sería la definición de la clase Empleado, clase base para otras que heredarán de ella. En este caso, para estrategia de tabla única, la podemos definir como abstracta para impedir crear instancias y obligar a que las instancias sean de las clases hijas. No es obligatorio hacerlo así, podría darse el caso de que nos interesara también tener instancias de la clase padre sin especificar el tipo de empleado.

```
@Entity
//@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipoEmpleado", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="empl")
public abstract class Empleado {
   ...
}
```

Nótese que la línea donde está definido el tipo de mapeo para la herencia está comentada. Esto es porque no es necesaria, ya que SINGLE_TABLE es el valor por defecto.

Las subclases de la jerarquía se definen igual que en Java estándar (recordemos que las entidades son clases Java normales) con la palabra clave *extends*. Lo único que hay que añadir es el valor en la columna discriminante que se le asigna a esta clase. Para ello se utiliza la anotación *DiscriminatorValue* y su atributo *value*. Por ejemplo, si el EmpleadoBecario va a tener como valor la cadena *beca*, hay que indicar:

```
@Entity
@DiscriminatorValue(value="beca")
```

Incluso en el caso de estrategia de tabla única, los mapeados se han de incluir en el *hibernate.cfg.xml* no sólo para la clase base, sino también para las clases derivadas.

Por último, resaltar que una entidad puede heredar de otra entidad o de una clase que no sea una entidad. Si la clase padre es una entidad, también su estado se estado se guarda al persistir a la clase hija. Si la clase padre no es una entidad su estado no se guarda.

6. HQL

Para el caso de las búsquedas se utiliza el lenguaje *HQL* (Hibernate Query Language), muy similar al lenguaje SQL que se usa en las bases de datos relacionales, pero en este caso totalmente Orientado a Objetos, ya que en vez de trabajar con las tablas y sus columnas se trabaja con objetos y sus atributos.

Dejamos de lado, por el tiempo que requeriría, el uso de las llamadas *Named Queries*, (consultas con nombre, haciendo uso de anotaciones como *@NamedQuery*) de uso muy común en JPA e Hibernate.

Las consultas HQL son traducidas por Hibernate a consultas SQL convencionales. Aunque es posible utilizar sentencias SQL con Hibernate, se recomienda utilizar HQL para prevenir problemas de portabilidad en la base de datos y aprovechar las ventajas de cacheado propias de Hibernate.

Términos como SELECT, FROM, y WHERE, etc., no son sensibles al uso de mayúsculas o minúsculas, aunque sí lo sean propiedades como los nombres de las tablas y columnas. Veamos algún ejemplo inicial.

- Obtener un **objeto identificado por el id**

```
. . .
int id = . . .;
Cliente cliente = HibernateUtil.getCurrentSession().get(Cliente.class, id);
. . .
```

Para realizar una consulta usaremos el método *createQuery()*, al que se le pasará un String con la consulta. Por ejemplo:

```
Query q = session.createQuery("from Departamento");
```

¡Ojo!: Tenemos *Query* en *javax.persistence* (*q.getResultList()*), pero utilizaremos la de *org.hibernate.query*.

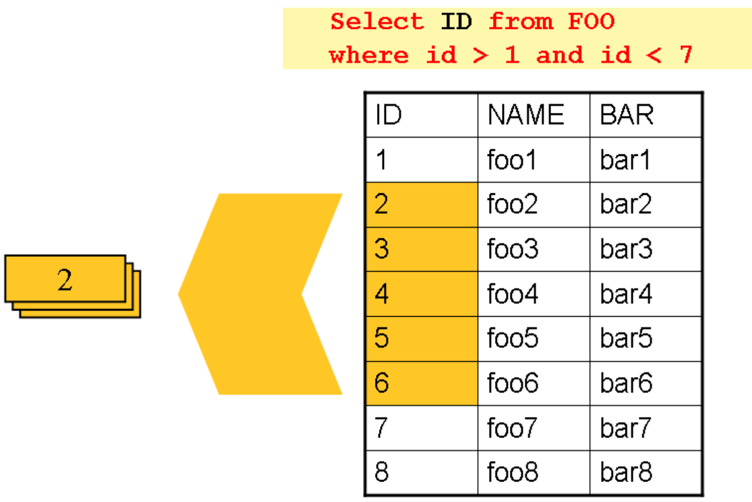
Para recuperar los datos usaremos el método *list()* (marcado como obsoleto) o *getResultList()*. También tenemos *getSingleList()* cuando se espera un único resultado.

```
List<Departamento> lista = q.getResultList();
```

O mediante *iterate()*.

```
Iterator iter = q.stream().iterator();
```

El primero de ellos, *list()*, retorna una colección con todos los resultados de la consulta incluyendo todas las entidades. Esto lo hace con una única operación sobre la base de datos, lo cual requiere que haya memoria suficiente para almacenar todos los objetos. Si la cantidad de resultados es extensa, el retraso del acceso a la base de datos será relevante.



Por su parte, *iterator()* retorna un iterador para recuperar los resultados de la consulta. Hibernate obtiene, en este caso, sólo los identificadores de las entidades y en cada llamada al método *Iterator.next()* obtiene la entidad completa. Esto supone una mucha mayor cantidad de accesos a la base de datos y, por lo tanto, mayor tiempo de procesamiento. La ventaja es que no tenemos todas las entidades cargadas en memoria simultáneamente. Se puede utilizar el método *setFetchSize()* para indicar la cantidad de resultados a recuperar en cada acceso a la base de datos, con cada *next()*.

No confundir *setFetchSize()* con *setMaxResults()*. Esta última es lo mismo que LIMIT en SQL: estás configurando la cantidad máxima de filas que quieres que se devuelvan, limita la cantidad de resultados que obtendrá la consulta.

setFetchSize() tiene que ver con la optimización, que puede cambiar la forma en que Hibernate envía los resultados (en fragmentos de diferentes tamaños). Le dice al controlador jdbc cuántas filas devolver en un fragmento, para consultas grandes. Por ejemplo, si hemos de obtener 1000 filas podemos configurar *setFethSize()* en 100, y la base de datos devolverá 100, luego otras 100, y así sucesivamente. El problema puede estar en que no está implementado por todos los controladores de bases de datos.

6.1. Cláusula FROM

Usaremos la cláusula FROM cuando queramos cargar una cantidad de objetos persistentes en memoria. Veamos un ejemplo.

```
String hql = "FROM Empleado";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

Si necesitamos especificar la clase con su paquete sería similar.

```
String hql = "FROM com.paquete.subpaquete.Empleado";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

6.2. Cláusula AS

La cláusula AS puede ser utilizada para asignar alias a las clases, especialmente en sentencias largas. El ejemplo anterior podría ser:

```
String hql = "FROM Empleado AS E";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

El uso de AS es opcional; también podemos especificar el alias directamente a continuación del nombre de la clase:

```
String hql = "FROM Empleado E";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

6.3. Cláusula SELECT

La cláusula SELECT permite obtener unas pocas propiedades de los objetos en lugar de los mismos objetos completos. El siguiente ejemplo muestra cómo obtener sólo el nombre de cada empleado.

```
String hql = "SELECT E.firstName FROM Empleado E";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

Es importante destacar aquí que *Empleado.firstName* es un atributo de la clase *Empleado*, no una columna de la tabla "empleados".

6.4. Cláusula WHERE

Esta cláusula la utilizamos para limitar la consulta a unos objetos específicos, veamos un ejemplo. Si el criterio especificado nos puede devolver más de un objeto:

```
String ciudad = . . .;
String hql = "FROM Cliente c WHERE c.ciudad = 'Petrer'";
Query query = session.createQuery(hql);
List<Cliente> clientes = (List<Cliente>) query.list();
```

Por contra, en el caso de retorno de un único valor usaríamos *uniqueResult()*.

```
String hql = "FROM Empleado E WHERE E.id = 10";
Query query = session.createQuery(hql);
Empleado empleado = (Empleado) query.uniqueResult();
```

6.5. Cláusula ORDER BY

Para ordenar los resultados de la consulta por un determinado valor. Podemos especificar, además, si queremos que sea ascendente ASC o descendente DESC .

```
String hql = "FROM Empleado E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

Si queremos ordenar por más de una propiedad, bastará con añadir las propiedades adicionales al final separando por comas.

```
String hql = "FROM Empleado E WHERE E.id > 10 " +
"ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List<Empleado> results = query.list();
```

6.6. Cláusula GROUP BY

Nos permitirá agrupar los resultados en base al valor de un atributo y, típicamente, utilizar el resultado para incluir un valor agregado.

```
String hql = "SELECT SUM(E.salary), E.firstName FROM Empleado E " +
"GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

6.7. Utilizando parámetros con nombre

Hibernate soporta dar nombre a los parámetros utilizados en las consultas. Esto facilita las consultas que aceptan valores de entrada por parte del usuario previniendo de ataques de inyección de código. Veamos un ejemplo.

```
String hql = "FROM Empleado E WHERE E.id = :Empleado_id";
Query query = session.createQuery(hql);
query.setParameter("Empleado_id",10);
Empleado empleado = query.uniqueResult();
```

6.8. Consultas sobre más de una clase

Si queremos recuperar los datos de una consulta en la que intervienen varias entidades podemos utilizar la clase *Object*. Los resultados obtenidos con *iterate()* los recibimos en un array de objetos, donde el primer elemento se corresponde con la primera clase a la derecha del FROM y el resto en secuencia.

```
String hql = "FROM Empleado E, Departamento D WHERE\
                         E.departamento.deptNo = D.deptNo order by E.apellido";
Query q = session.createQuery(hql);
Iterator it =  q.iterate();
while (it.hasNext()) {
    Object par[] = (Object[]) it.next();
    Empleado emp = (Empleado) par[0];
    Departamento dep = (Departamento) par[1];
    System.out.println(emp);
    System.out.println(dep);
}
```

6.9. Cláusula UPDATE

Desde la versión 3 de Hibernate, los borrados y actualizaciones se pueden hacer con la interfaz *Query* y su método *executeUpdate*.

```
String hql = "UPDATE Empleado set salary = :salary "
"WHERE id = :Empleado_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("Empleado_id", 10);
int result = query.executeUpdate();
System.out.println("Filas afectadas: " + result);
```

6.10. Cláusula DELETE

```
String hql = "DELETE FROM Empleado " +
"WHERE id = :Empleado_id";
Query query = session.createQuery(hql);
query.setParameter("Empleado_id", 10);
int result = query.executeUpdate();
System.out.println("Filas afectadas: " + result);
```

6.11. Cláusula INSERT

HQL soporta esta cláusula INSERT INTO sólo cuando los valores pueden ser insertados desde un objeto en otro.

```
String hql = "INSERT INTO Empleado(firstName, lastName, salary)" +
"SELECT firstName, lastName, salary FROM old_Empleado";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Filas afectadas: " + result);
```

Es importante resaltar que HQL no soporta la sintaxis directa INSERT INTO ... VALUES como el SQL estándar. En HQL, las inserciones sólo las podemos hacer seleccionando datos de otra entidad, con la sintaxis:

INSERT INTO EntidadDestino (prop1, prop2, ...) SELECT propX, propY FROM EntidadOrigen WHERE ...

6.12. Consultas con SQL

También es posible lanzar **consultas directamente en lenguaje SQL**, trabajando entonces directamente con las tablas y campos de la base de datos. Para ello no utilizamos *Query* sino *SQLQuery*.

```
String ciudad = "Petrer";
SQLQuery sqlQuery = HibernateUtil.getCurrentSession().
createSQLQuery("SELECT nombre, apellidos FROM clientes WHERE ciudad = :ciudad");
query.setParameter("ciudad", ciudad);
List<Object[]> resultado = sqlQuery.list();
for (Object[] objeto : resultado) {
        Cliente cliente = new Cliente();
        cliente.setNombre(objeto[0].getNombre());
        cliente.setApellidos(objeto[1].getApellidos());
        System.out.println(cliente);
}
```

6.13. Consultas con CriteriaQuery

El método *createQuery(String)* de *Session* ha sido marcado como obsoleto en favor del uso de *createQuery(CriteriaQuery)*. Veamos 2 ejemplos de consultas con esta metodología.

```
// Consulta primera: mostrar todos los alumnos de una determinada edad
    int edadAlumno = 25;
    CriteriaBuilder cb = session.getCriteriaBuilder();

    CriteriaQuery<Alumno> cq = cb.createQuery(Alumno.class);
    Root<Alumno> r = cq.from(Alumno.class);
    cq.where(cb.equal(r.get("edad"),edadAlumno));
    Query<Alumno> q = session.createQuery(cq);
    List<Alumno> lista = q.list();
    System.out.println("Los alumnos de " + edadAlumno + " años son:");
    for (Alumno al: lista)
        System.out.println(al.getNom());

// Consulta segunda: cuántos alumnos tienen esa misma edad
    CriteriaQuery<Long> cq2 = cb.createQuery(Long.class);
    Root<Alumno> r2 = cq2.from(Alumno.class);
    cq2.select(cb.countDistinct(r2)).where(cb.equal(r2.get("edad"), edadAlumno));
    Query<Long> q2 = session.createQuery(cq2);
    Long result = q2.uniqueResult();
    System.out.println("Número de alumnos con " + edadAlumno + " años: " + result);
```

En general, las formas vistas inicialmente para JPQL y HQL se suele recomendar para las consultas estáticas, mientras que usando *Criteria* se hacen más fáciles las consultas dinámicas. Pero ¿cuáles son dinámicas o estáticas? Estáticas son aquellas en las que siempre tenemos las mismas condiciones. Por ejemplo, el precio de un producto ha de ser mayor que un determinado valor; no siempre el mismo, pero sí que siempre comprobaremos el mismo parámetro: el precio. Por otro lado, las dinámicas son aquellas en las que los criterios dependen de los parámetros que recibamos. Por ejemplo, nos pueden pasar el nombre y buscamos por nombre, pero si nos pasan el apellido, buscamos por apellido, y si nos pasan ambos, buscamos por nombre y apellido.

¿Es mejor entonces resolver las consultas con *Criteria*? La decisión correcta dependerá de las circunstancias, ya no sólo de cada proyecto, sino de cada caso. Lo importante: saber hacer consultas de las dos formas y, conociendo esos pros y contras, poder decidir. No se debe decidir simplemente porque me resulta más fácil una que la otra. También se ha de tener en cuenta que, en cuanto a seguridad, con *Criteria* se elimina la posibilidad de inyección de SQL.

7. COMPARATIVA JDBC-JPA

Antes de acabar el estudio de Hibernate hemos de recordar que se trata de una herramienta que, a bajo nivel, utiliza *JDBC*. Ello hace que, realmente, cuando interesa el máximo de eficiencia nos pueda interesar trabajar directamente con JDBC. Los beneficios de simplicidad que nos ofrece Hibernate, u otras implementaciones de mapeado, tienen un coste y es que el rendimiento se degrada debido a todas las conversiones que se han de hacer para convertir las *Entity* en *Querys* y los *ResultSet* pasarlos a clases. Resumiendo las ventajas e inconvenientes podríamos decir:

Ventajas de JPA, Hibernate:

1. Nos permite desarrollar mucho más rápido.
2. Permite trabajar con la base de datos por medio de entidades en vez de sentencias SQL.
3. Nos ofrece un paradigma 100% orientado a objetos.
4. Elimina errores en tiempo de ejecución.
5. Mejora el mantenimiento del software.

Desventajas:

1. No ofrece toda la funcionalidad que ofrecería lanzar consultas nativas.
2. El rendimiento es mucho más bajo que realizar las consultas por JBDC.
3. Puede representar una curva de aprendizaje más grande.
