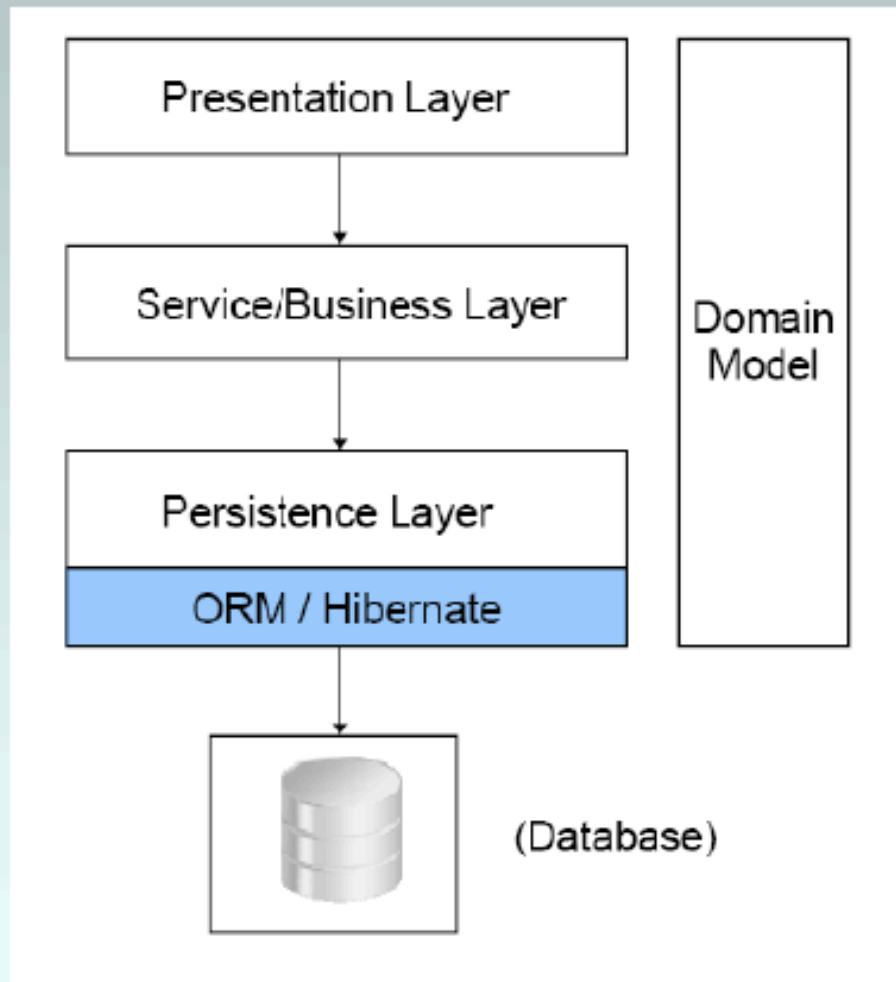


Visión general de Hibernate

- Hibernate es una solución para gestionar de manera transparente la persistencia en entornos orientados a objeto.
- Permite en tiempo de ejecución transformar objetos en tuplas y tuplas en objetos.
- Solución independiente de la BD utilizada.
- Utiliza archivos XML:
 - Para definir la transformación de clases a tablas (archivos de "mapping").
 - Para indicar qué base de datos se utilizará (archivo de configuración)



Visión general de Hibernate: estructura de las clases persistentes

- Para que una clase pueda ser persistente debe tener:

- Constructor por defecto.
- Métodos getXXX, setXXX para cada atributo.
- Un atributo que actúe como identificador.

```
public class Mensaje {  
  
    private int id;  
    private String text;  
    private Mensaje SiguienteMensaje;  
    Mensaje() {}  
    public Mensaje(String text) {  
        this.text = text;  
    }  
    public int getId() {  
        return id;  
    }  
    @SuppressWarnings("unused")  
    private void setId(int id) {  
        this.id = id;  
    }  
    public String getText() {  
        return text;  
    }  
    public void setText(String text) {  
        this.text = text;  
    }  
    public Mensaje getSiguienteMensaje() {  
        return SiguienteMensaje;  
    }  
    public void setSiguienteMensaje(Mensaje  
    SiguienteMensaje) {  
        this.SiguienteMensaje = SiguienteMensaje;  
    }  
}
```

Archivo de configuración

```
<?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.hsqldb.jdbcDriver</property>
    <property
name="hibernate.connection.url">jdbc:hsqldb:hsqldb://localhost/mensajes</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</property>
    <mapping resource = "Mensaje.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Configuración de la conexión JDBC

Variante de SQL

Archivos de mapping

Visión general de Hibernate: mappings

- Los archivos de mapping definen cómo se transforman las clases del modelo en tablas, en particular:
 - Equivalencia atributo – columna.
 - Cómo se generan las claves primarias, para las inserciones de registros (generator class).
 - Cómo se relacionan mediante claves ajenas con otras tablas.

Interface Session

- La interfaz Session permite recuperar y almacenar objetos en la base de datos a través de una comunicación con el motor de Hibernate.

- Un objeto sesión se obtiene de una factoría de sesiones:

```
session = UtilidadHibernate.getSessionFactory().openSession();
```

- Proporciona una caché (contexto de persistencia) que evita interacciones innecesarias con la base de datos. Cuando ejecutamos las operaciones: *find()*, *update()*, *save()*, *saveOrUpdate()*, *get()*, *delete()* o cualquier otra operación de la interfaz Session, estamos interactuando de manera transparente con la caché de Hibernate. Las operaciones no se realizan directamente sobre la base de datos se almacenan en la caché.

Interface Session

- La interfaz Session permite recuperar y almacenar objetos en la base de datos a través de una comunicación con el motor de Hibernate.

- Un objeto sesión se obtiene de una factoría de sesiones:

```
session = UtilidadHibernate.getSessionFactory().openSession();
```

- Proporciona una caché (contexto de persistencia) que evita interacciones innecesarias con la base de datos. Cuando ejecutamos las operaciones: *find()*, *update()*, *save()*, *saveOrUpdate()*, *get()*, *delete()* o cualquier otra operación de la interfaz Session, estamos interactuando de manera transparente con la caché de Hibernate. Las operaciones no se realizan directamente sobre la base de datos se almacenan en la caché.

Interface Session: Operaciones

Hacer persistente un objeto transitorio.

```
Mensaje Mensaje = new Mensaje("Hola a  
todo(a)s, primer mensaje.");  
sesion.save(Mensaje);
```

Obtener un objeto si se sabe que existe.

```
Mensaje mensaje = (Mensaje)  
sesion.get(Mensaje.class, ID);
```

Obtener un objeto si no se está seguro de que existe.

```
Mensaje mensaje = (Mensaje)  
terceraSession.load(Mensaje.class,  
ID);
```

Borrar un objeto

```
sesion.delete(mensaje);
```

Actualizar

```
sesion.update(mensaje);
```

Salvar o actualizar

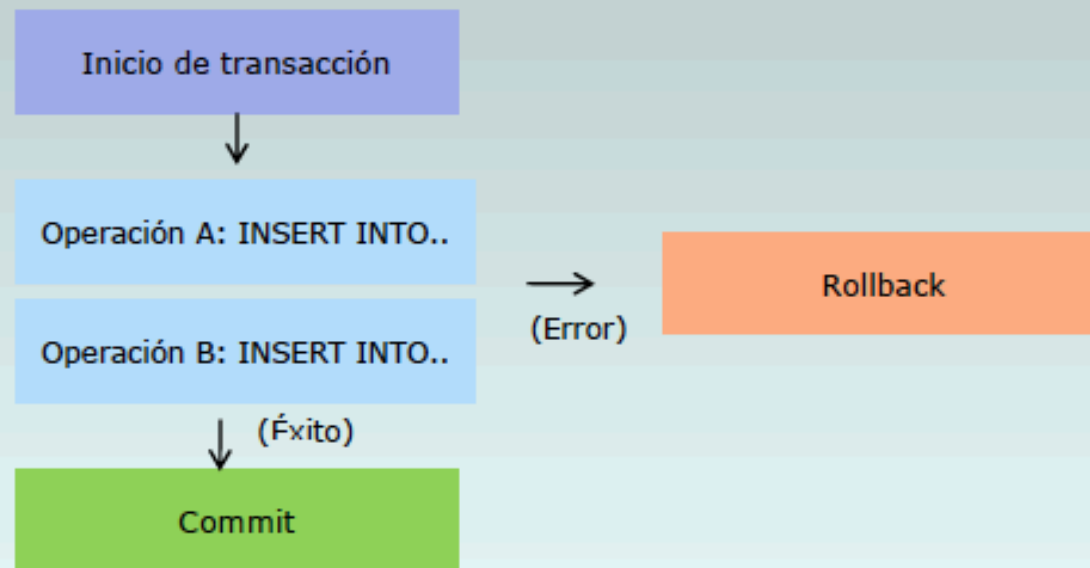
```
sesion.saveOrUpdate(mensaje);
```

Sincronizar el contexto de persistencia con la BD.

```
Sesion.flush()
```

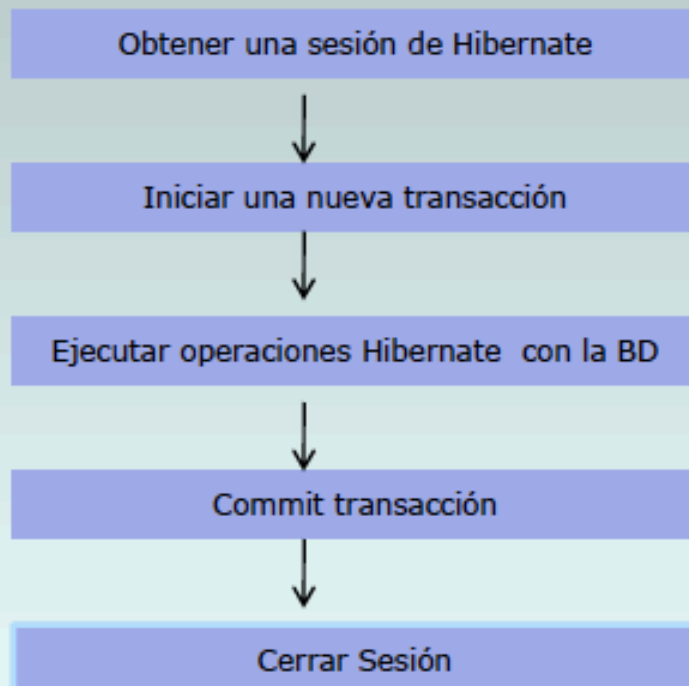

Transacción

- Una transacción es un conjunto de operaciones sobre una base de datos que terminan en un commit (éxito) o bien en un rollback (fallo).



Transacciones en Hibernate

El patrón común es utilizar una sesión por transacción.



```
Session =
    UtilidadHibernate.getSessionFactory().
        openSession();
Transaction tx = Session.beginTransaction();

try {
    Session.save(Mensaje);
    tx.commit();
} catch( RuntimeException e)
{
    if (tx != null)
    { tx.rollback(); throw e;
    }
}
finally {
    Session.close();
}
```

Visión general de Hibernate: acceso a datos

- Para recuperar un objeto de la base de datos:

```
Session terceraSession =
    UtilidadHibernate.getSessionFactory().openSession();
Transaction terceraTransaction =
    terceraSession.beginTransaction();
int ID = 1;
Mensaje mensaje = (Mensaje)
    terceraSession.get(Mensaje.class, ID);
    mensaje.setSiguienteMensaje(new Mensaje("Hola,
después de hola a todo(a)s"));
    terceraTransaction.commit();
    terceraSession.close();
```

Convertir en persistente un objeto

- Dentro de una sesión para convertir un objeto transitorio en persistente se utiliza el método `save()`.

```
User user = new User();  
user.getName().setFirstname("John");  
user.getName().setLastname("Doe");  
Transaction tx = session.beginTransaction();  
session.save(user);  
tx.commit(); // session.flush();  
session.close();
```

- La sincronización con la base de datos se realiza al ejecutar el método `commit()` de la transacción.
- En ese punto Hibernate obtiene una conexión JDBC ejecuta un sentencia SQL de inserción y al cerrar la sesión la conexión también se cierra.
- Si la transacción falla el `rollback` se realiza a nivel de base de datos no a nivel de objetos en memoria.

Objetos transitorios

- Los objetos instanciados mediante `new` no son por defecto persistentes.
- Su estado es transitorio, es decir no están asociados con ninguna tupla de la base de datos.
- Su estado desaparece (vía el mecanismo java de recoge basura) cuando dejan de estar referenciados.
- Los objetos referenciados por objetos transitorios son también considerados transitorios.
- Para convertirlos en persistentes se utiliza el método `save()` del gestor de persistencia de Hibernate.
- No existe ningún mecanismo de rollback para objetos transitorios, ya que no son transaccionales.

Obtener un objeto persistente

- La sesión se utiliza también para obtener objetos persistentes, aunque Hibernate posee un lenguaje de consultas (explicado después), se puede emplear el método `get()` de sesión para obtener un objeto mediante su clave.

```
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
tx.commit();
session.close();
```

- El método `get()` busca el objeto cuyo id sea 1234. Si no existe ninguno con esa clave devuelve null.
- Una vez cerrada la sesión el objeto pasa al estado separado.

Actualizar un objeto persistente

- Cualquier objeto persistente se encuentra asociado con una sesión (contexto de persistencia).
- Si se modifica su estado mediante el mecanismo llamado "automatic dirty checking" Hibernate lleva a cabo la sincronización con la base de datos dentro del contexto actual de persistencia.

```
Transaction tx = session.beginTransaction();  
int userID = 1234;  
User user = (User) session.get(User.class, new Long(userID));  
user.setPassword("secret");  
tx.commit();  
session.close();
```

- Cuando se ejecuta `commit()` se produce la sincronización.
- Obviamente al cerrar la sesión el objeto pasa al estado separado.

Convertir un objeto persistente en "removed".

- Los objetos persistentes se pueden convertir en transitorios mediante el método `delete()`.

```
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
session.delete(user);
tx.commit();
session.close();
```

- La sentencia SQL de borrado se ejecuta cuando la sesión se sincroniza con la base de datos mediante el `commit`.
- Al cerrar la sesión el objeto se convierte en transitorio, si desaparecen las referencias al mismo, el mecanismo de liberación de memoria lo destruye.

Obtención de objetos persistentes

- Para extraer objetos desde la base de datos en Hibernate se pueden utilizar las siguientes opciones:
 - Navegar por el grafo de objetos: Una vez cargado un objeto y dentro de una sesión se puede acceder a sus propiedades y métodos. Esto origina que el gestor de persistencia cargue automáticamente los nodos (objetos) al intentar acceder por navegación a alguna de sus propiedades.
 - `unPedido.Cliente.getNombre();`
 - Obtenerlos mediante el identificador (id), es el método más conveniente cuando el identificador se conoce.
 - Utilizando el lenguaje de consultas de Hibernate (HQL) una versión orientada a objetos de SQL.
 - Utilizando el lenguaje SQL nativo de la base de datos, donde Hibernate realiza la conversión de los registros (JDBC result set) en grafos de objetos persistentes.

Lenguaje de consultas de Hibernate (HQL)

- HSQL se encarga de la traducción de consultas realizadas sobre objetos a consultas realizadas sobre registros y a la inversa, convierte los registros en objetos.
- Sobre el diagrama de clases de diseño, la consulta HSQL:



FROM Persona P where P.categoria.id = 3

- Se traduce en la sentencia SQL:
Select ID, Nombre, Apellidos from Personas where categoria =3



Query: definición de consultas

- La interface Query define varios métodos para controlar la ejecución de consultas.
- Para crear una consulta, en el contexto de una sesión, se utilizan los métodos `createQuery()` y `createSQLQuery`:

- `createQuery()` se emplea para consultas HQL:

```
Query hqlQuery = session.createQuery("from Usuario");
```

- `createSQLQuery` se usa para consultas en SQL nativo:

```
Query sqlQuery = session.createSQLQuery(  
    "select {u.*} from USUARIOS {u}", "u", Usuario.class);
```

O de manera alternativa:

```
Query sqlQuery = session.createSQLQuery("select {usuario.*} from USUARIOS  
{usuario}").addEntity("usuario", Usuario.class);
```

- En ambos casos se obtiene una instancia del objeto Query pero no se ejecuta ninguna consulta sobre la base de datos.

Ejemplo Mapping:

Transformaciones de una clase

■ Clase sin relaciones

```
public class Contacto implements
    Serializable {
    private int id;
    private String Nombre;
    private String Apellido;
    private String email;

    public Contacto() {
        //constructor por defecto
    }

    public Contacto(String nombre, String
        apellido, String email, int id) {
        Nombre = nombre;
        Apellido = apellido;
        this.email = email;
        this.id = id;
    }

    public String getApellido() {
        return Apellido;
    }
}
```

```
public void setApellido(String
    apellido) {
    Apellido = apellido;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public int getId() {
    return id;
}

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

public String getNombre() {
    return Nombre;
}

public void setNombre(String nombre)
    {
        Nombre = nombre;
    }
}
```

Ejemplo Mapping: Transformaciones de una clase

- Archivo de correspondencia

```
<hibernate-mapping>
  <class name="negocio.Contacto" table="CONTACTOS">
    <id name="id" type="integer" column="ID">
      <generator class="assigned"></generator>
    </id>
    <property name="Nombre">
      <column name="Nombre" length="100" />
    </property>
    <property name="Apellido">
      <column name="Apellido" length="100"/>
    </property>
    <property name="email">
      <column name="email" length="200"/>
    </property>
  </class>
</hibernate-mapping>
```

Relaciones y colecciones

- La multiplicidad en el modelo conceptual de las relaciones de asociación /agregación puede ser:
 - 1:1 (one-to-one)
 - N:1 (many-to-one)
 - 1:N (one-to-many)
 - N:M (many-to-many)
- En el caso de relaciones con multiplicidades muchos, surge el concepto de colección en Hibernate.
- Hibernate contiene las siguientes colecciones:

Colección Hibernate	Colección Java	Descripción
set	java.util.Set	Colección no ordenada de valores de objetos sin repeticiones.
map	java.util.Map	Colección de pares clave/valor.
list	java.util.List	Colección ordenada de valores de objetos, admite repetidos.
bag	java.util.List	Colección no ordenada de valores de objetos, admite repetidos.
array	-	Colección indexada, con repetidos, de valores de objetos.
primitive-array	-	Colección indexada, con repetidos, de valores primitivos.
idbag	java.util.List	Colección muchos a muchos no ordenada, con repetidos, utilizando una clave adicional.

Ejemplo: Ficheros de 'mapping'

- Inicialmente vamos a definir relaciones unidireccionales
- *File | New | Other | Hibernate XML Mapping File*
- 'Mapping' *Pelicula*:

```
<class name="Pelicula" table="PELICULA" schema="PUBLIC">
  <id name="idpelicula" type="int">
    <column name="IDPELICULA" />
    <generator class="native" />
  </id>
  <property name="titulo" column="TITULO"/>
  <property name="fecha" column="FECHA" />
  <property name="precio" column="PRECIO" />
  <property name="argumento" column="ARGUMENTO"/>
  <many-to-one name="director" class="Director" >
    <column name="IDDIRECTOR" />
  </many-to-one>
  <many-to-one name="genero" class="Genero" >
    <column name="IDGENERO" />
  </many-to-one>
</class>
```

Estrategia de generación de ids: escoge la mejor según el SGBD. En este caso es equivalente a *increment*

Relación many-to-one entre *Pelicula* y *Director* mediante la clave ajena *IDDIRECTOR*

Ejemplo Mapping: Transformaciones de una clase

- Archivo de correspondencia

```
<hibernate-mapping>
```

```
  <class name="negocio.Contacto" table="CONTACTOS">  
    <id name="id" type="integer" column="ID">  
      <generator class="assigned"></generator>
```

```
    </id>
```

```
    <property name="Nombre">
```

```
      <column name="Nombre" length="100" />
```

```
    </property>
```

```
    <property name="Apellido">
```

```
      <column name="Apellido" length="100"/>
```

```
    </property>
```

```
    <property name="email">
```

```
      <column name="email" length="200"/>
```

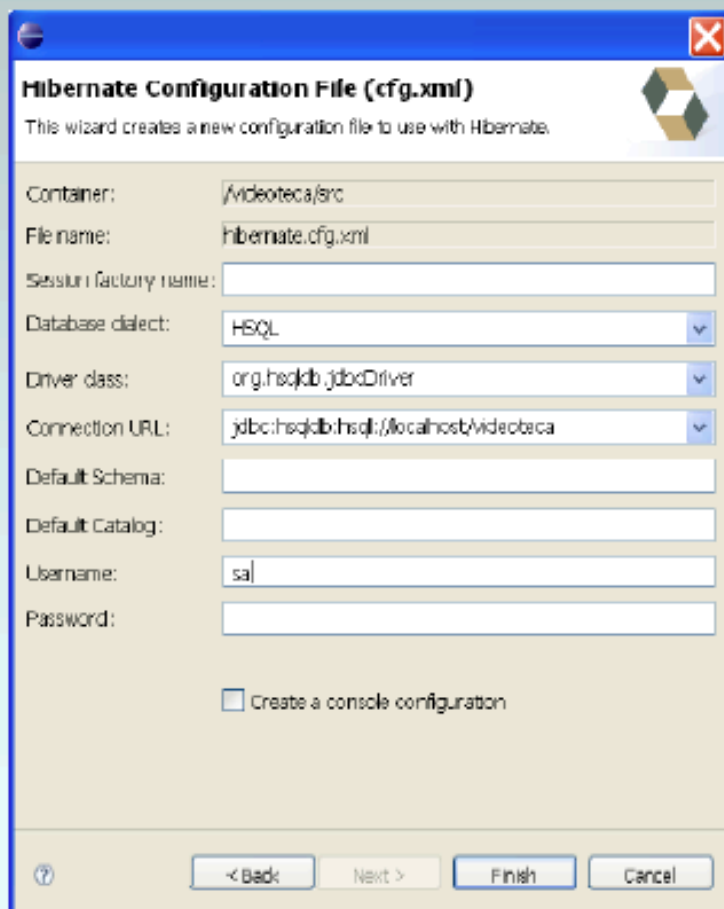
```
    </property>
```

```
  </class>
```

```
</hibernate-mapping>
```

Ejemplo: Fichero de configuración de Hibernate

- Desde la carpeta src del proyecto:
- New | Other | Hibernate -> Hibernate Configuration File



```
<?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.hsqldb.jdbcDriver</pr
operty>
        <property
name="hibernate.connection.url">jdbc:hsqldb:hsqldb://localhost/video
teca</property>
        <property
name="hibernate.connection.username">sa</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</proper
ty>
        <mapping resource="Negocio/Pelicula.hbm.xml"/>
        <mapping resource="Negocio/Director.hbm.xml"/>
        <mapping resource="Negocio/Genero.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Añadir a
mano

Ejemplo Paquete AccesoAdatos

- Crear la clase PeliculaDAOHibernateImp que implementa a la interfaz IPeliculaDAO:

```
public class PeliculaDAOHibernateImp implements IPeliculaDAO {

    private Session session = null; // Hibernate Session
    private Transaction tx = null; // Hibernate Transaction

    public void createMovie(Pelicula p) {
        // TODO Auto-generated method stub
        try {
            session = UtilidadHibernate.getSessionFactory().openSession();
            tx = session.beginTransaction();
            session.save(p);
            tx.commit();
        } catch (HibernateException e) { rollback(); throw e;
            } finally {
                session.close();
            }
    }
}
```

Paquete AccesoAdatos: Dao Película

- Las operaciones DAO utilizan a Session, Query y Transaction:

```
private void rollback()
{
    try {
        if (tx != null) {
            tx.rollback();
        }
    } catch (HibernateException ignored) {
        // No se puede hacer rollback de la transacción;
    }
}
```

- El resto de operaciones tienen el mismo patrón:
 - Abrir sesión, iniciar transacción
 - Llamar a la correspondiente operación de sesion (update, delete, etc.), o bien utilizar el lenguaje de consultas de Hibernate.
 - Commit sobre la transacción.
 - Cerrar la sesión.

Paquete AccesoAdatos: Dao Película

- Para borrar una película por id.

```
public void deleteMovie(int id) throws PeliculaNoEncontradaExcepcion
{
    // TODO Auto-generated method stub
    try {
        session = UtilidadHibernate.getSessionFactory().openSession();
        tx = session.beginTransaction();
        Object pelicula = session.load(Pelicula.class, id);
        session.delete(pelicula);
        tx.commit();
    } catch (HibernateException e) {
        rollback();
        throw new PeliculaNoEncontradaExcepcion();
    }
    finally {
        session.close();
    }
}
```

Paquete AccesoAdatos: Dao Película

- O bien pasando el objeto a borrar.

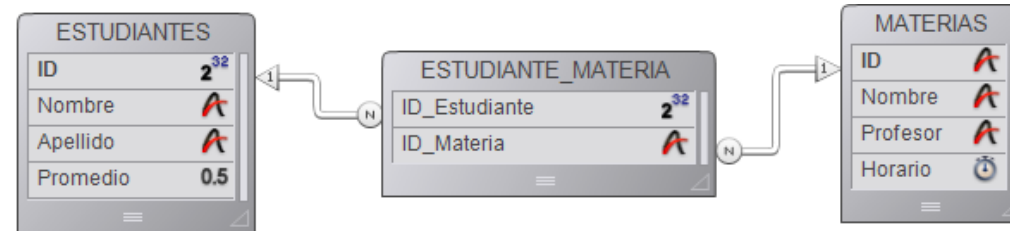
```
public void deleteMovie(Pelicula p) throws PeliculaNoEncontradaExcepcion
{
    // TODO Auto-generated method stub
    try {
        session = UtilidadHibernate.getSessionFactory().openSession();
        tx = session.beginTransaction();
        session.delete(p);
        tx.commit();
    } catch (HibernateException e) {
        rollback(); throw new PeliculaNoEncontradaExcepcion();
    }
    finally {
        session.close();
    }
}
```

Paquete AccesoAdatos: Dao Película

- Para obtener todas las películas:

```
public ArrayList findPelículas() {  
    // TODO Auto-generated method stub  
    try {  
        session = UtilidadHibernate.getSessionFactory().openSession();  
        session.beginTransaction();  
        ArrayList ListaPelículas = (ArrayList) session.createQuery("from  
Película").list();  
        session.getTransaction().commit();  
        return ListaPelículas;  
    }  
    catch (HibernateException e) {  
        rollback(); throw e;  
    }  
    finally {  
        session.close();  
    }  
}
```


RELACIONES MUCHOS A MUCHOS



```
public class Materia
{
    private long id;
    private String nombre;

    public Materia()
    {
    }

    public long getId()
    {
        return id;
    }

    protected void setId(long id)
    {
        this.id = id;
    }
}
```

```
    public String getNombre()
    {
        return nombre;
    }

    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }
}
```

```
public class Estudiante
{
    private long id;
    private String nombre;
    private List<Materia> materias = new ArrayList<Materia>();

    public Estudiante()
    {
    }

    public long getId()
    {
        return id;
    }
}
```

```
protected void setId(long id)

{
    this.id = id;
}

public List<Materia> getMaterias()
{
    return materias;
}

public void setMaterias(List<Materia> materias)
{
    this.materias = materias;
}

public String getNombre()
{
    return nombre;
}

public void setNombre(String nombre)
{
    this.nombre = nombre;
}
```

```
public void addMateria(Materia materia)

    {
        this.materias.add(materia);
    }
}
```

Una vez creados los archivos de mapeo, representamos las relaciones **muchos a muchos** usando el elemento acorde al tipo de colección que estemos usando (**list**, **set**, **map**, **array**, y **primitive-array**).

Para mapear una lista usamos el elemento "<list>". En este elemento indicamos cuál es el nombre del atributo, dentro de la clase **Estudiante**, que representa la relación. En este caso el atributo se llama "**materias**". También aquí indicamos cuáles operaciones queremos que se realicen en cascada. En este caso queremos que **todas** las operaciones de **guardar**, **actualizar** y **eliminar** que ocurran en el padre sean pasadas a la colección, o sea que cuando **guardemos**, **actualicemos**, o **eliminemos** un **Estudiante**, las operaciones pasen también a todas sus **Materias** relacionadas, por lo que usamos el valor "**all**".

En las relaciones **muchos a muchos**, igual que en las relaciones **uno a muchos**, existen dos estilos de cascada especiales llamados "**delete-orphan**" y "**all-delete-orphan**" (que solo pueden usarse con archivos de mapeo) los cuales se encargan de que, en el caso de que se elimine el objeto padre ("**Estudiante**"), todos los objetos hijos ("**Materia**") serán eliminados de la base de datos.

Adicionalmente "**all-delete-orphan**" se encarga de que todas las otras operaciones que mencionamos antes (**guardar**, **actualizar**, y **eliminar**) también sean realizados en cascada, por lo que usaremos este valor.

Para las relaciones de muchos a muchos se usa una tabla de unión o tabla join, para mantener los datos de que objetos de la entidad Estudiante están relacionados con los de la entidad Materia

En este caso debemos especificar cuál será el nombre de esta tabla de unión, usando el atributo "**table**" del elemento `<list>`. Por lo que este elemento queda, por el momento, así:

```
<list name="materias" table="ESTUDIANTES_MATERIAS" cascade="all-delete-orphan">  
</list>
```

La tabla de unión generada ("**ESTUDIANTES_MATERIAS**") tendrá como columnas "**id**", que es la llave foránea de la tabla estudiantes, y una segunda columna llamada "**elt**" (elt significa element, que es el nombre que da por default hibernate si no especificamos uno) que es la llave foránea de la tabla "**materias**".

Ahora indicamos cuál será el valor que se usará como llave foránea para relacionar las **Materias** con el **Estudiante**. Siempre, lo que usaremos será la llave primaria de la entidad que estamos mapeando (en este caso **Estudiante**), por lo que colocamos aquí el nombre de la columna que mantiene este valor, que en este caso es **"ID_ESTUDIANTE"**:

```
<key column="ID_ESTUDIANTE " />
```

Ahora bien, las listas son una estructura de datos con una característica única: **tienen un orden**. Esto significa que el orden en el que los elementos entran en la lista es importante e, internamente, se usa un índice para saber el orden de los elementos.

Cuando tratamos de almacenar estos datos nos interesa que en el momento que sean recuperados, los elementos de la lista estén en el mismo orden en el que los guardamos y es por esta razón que se debe usar una columna extra en la tabla de unión generada para guardar este índice (el cual comienza en cero). Para indicar el nombre que tendrá esta columna usamos el elemento **"index"** y colocamos en su atributo **"column"** el nombre que tendrá esta columna:

```
<index column="ORDEN" />
```

Las relaciones **muchos a muchos** las representamos usando el elemento **<many-to-many>**. Este elemento se coloca dentro del elemento **<list>** (o el elemento que esten usando para representar su relación) que acabo de explicar, y lo único que debemos indicarle de qué clase son las entidades que estamos guardando en la lista y cual es la columna que se usa para almacenar el id de esa entidad:

```
<many-to-many class="hibernate.relaciones.muchos.muchos.modelo.Materia"  
column="ID_MATERIA" />
```


El archivo de mapeo para la entidad `Estudiante` queda así:

```
<hibernate-mapping>
  <class      name="hibernate.relaciones.muchos.muchos.modelo.Estudiante"
table="ESTUDIANTES">
    <id name="id" column="ID_ESTUDIANTE">
        <generator class="identity" />
    </id>

    <property name="nombre" />

    <list name="materias" table="ESTUDIANTES_MATERIAS" cascade="all-
delete-orphan" >
        <key column="ID_ESTUDIANTE" />
        <list-index column="ORDEN" />
        <many-to-many
class="hibernate.relaciones.muchos.muchos.modelo.Materia"
column="ID_MATERIA" />
    </list>

  </class>
</hibernate-mapping>
```

En el siguiente código probamos la aplicación:

```
public static void main(String[] args)
{
    Estudiante estudiante1 = new Estudiante();
    estudiante1.setNombre("estudiante1");

    Materia material1 = new Materia();
    material1.setNombre("material1");
    Materia material2 = new Materia();
    material2.setNombre("material2");
    Materia material3 = new Materia();
    material3.setNombre("material3");

    estudiante1.addMateria(material1);
    estudiante1.addMateria(material2);
    estudiante1.addMateria(material3);

    Estudiante estudiante2 = new Estudiante();
    estudiante2.setNombre("estudiante2");

    Materia material4 = new Materia();
```

```
materia4.setNombre("materia4");  
Materia materia5 = new Materia();  
materia5.setNombre("materia5");  
Materia materia6 = new Materia();  
materia6.setNombre("materia6");
```

```
estudiante2.addMateria(materia4);  
estudiante2.addMateria(materia5);  
estudiante2.addMateria(materia6);
```

```
Session sesion = HibernateUtil.getSessionFactory().openSession();
```

```
sesion.beginTransaction();  
sesion.save(estudiante1);  
sesion.save(estudiante2);  
sesion.getTransaction().commit();  
sesion.close();
```

```
sesion = HibernateUtil.getSessionFactory().openSession();  
sesion.beginTransaction();  
sesion.delete(estudiante1);  
sesion.getTransaction().commit();  
sesion.close();
```

```
}
```