

# Hibernate



- Introducción
- Arquitectura
- Proceso de desarrollo
- Configuración
- Ejemplos
- Lenguajes de consulta
- Patrones de diseño
- Tips

## Hibernate - Introducción

- ¿Qué es Hibernate?
  - Es una herramienta de mapeo objeto/relacional (ORM) para ambientes Java.
  - Framework de persistencia, basado en objetos, para bases de datos relacionales.
- ¿Por qué surge?
  - Necesidad de persistir objetos Java en bases de datos relacionales.

## Hibernate

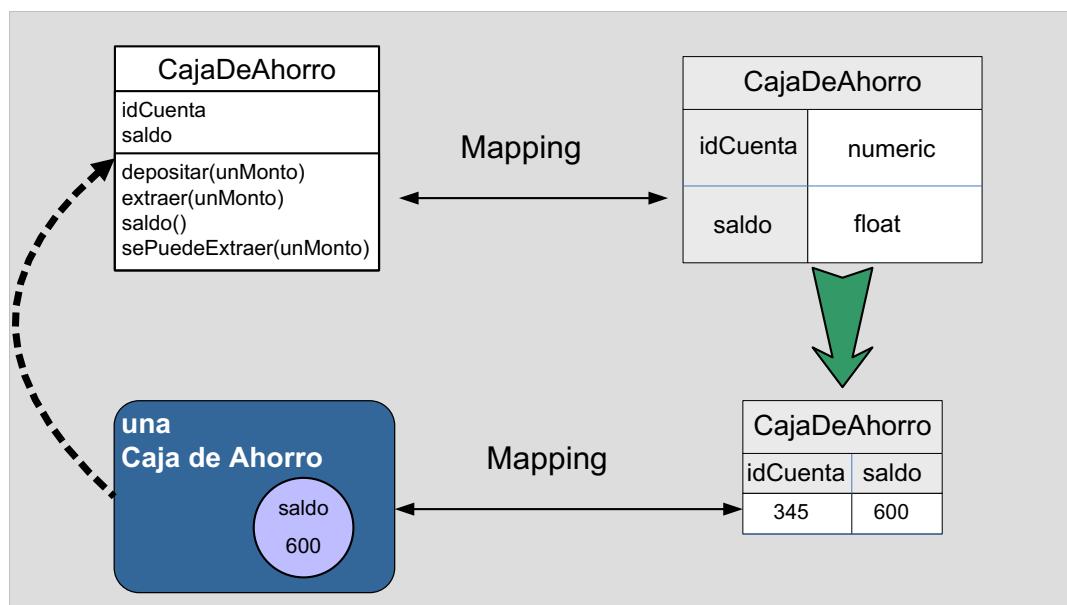
- ¿Qué características tiene?
  - Herencia
  - Polimorfismo
  - Relaciones uno a muchos, muchos a uno, uno a uno, muchos a muchos
  - Claves compuestas
  - Colecciones de datos
  - Cache
    - Varios cache providers
  - Transacciones
  - Lazy Initialization
  - HQL - SQL

## ORM - Object Relational Mapping

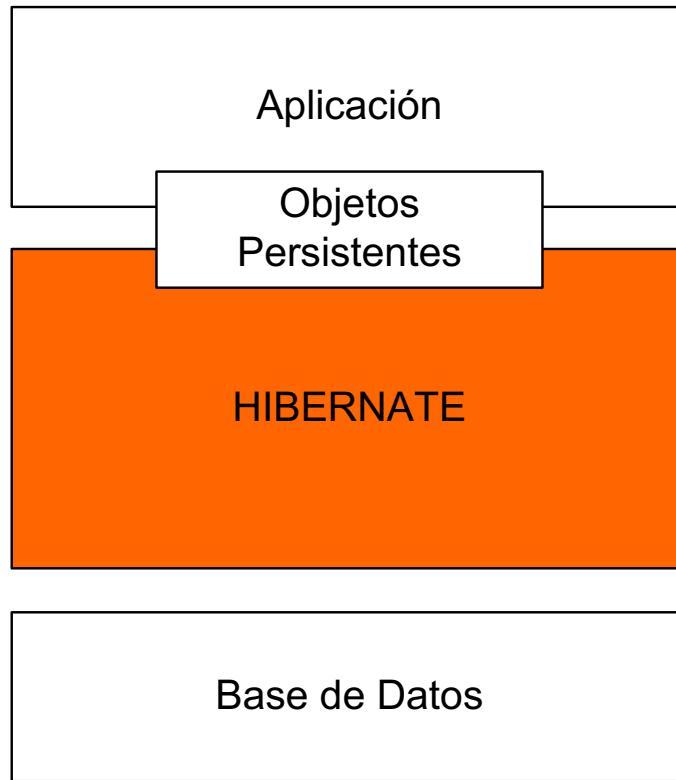
- ¿Qué es un mapping o mapeo?
  - Es un conjunto de reglas que establecen la forma en que los conceptos OO son llevados a conceptos en el modelo relacional y viceversa.
- ¿Cómo debe ser?
  - Idempotente.
  - Sin pérdida de información.
- ¿Cómo se establece el mapeo?
  - Hibernate utiliza comúnmente un archivo asociado a cada clase persistente **con extensión hbm.xml**; Ej. Consumidor.java -> Consumidor.hbm.xml.

# ORM - Object Relational Mapping

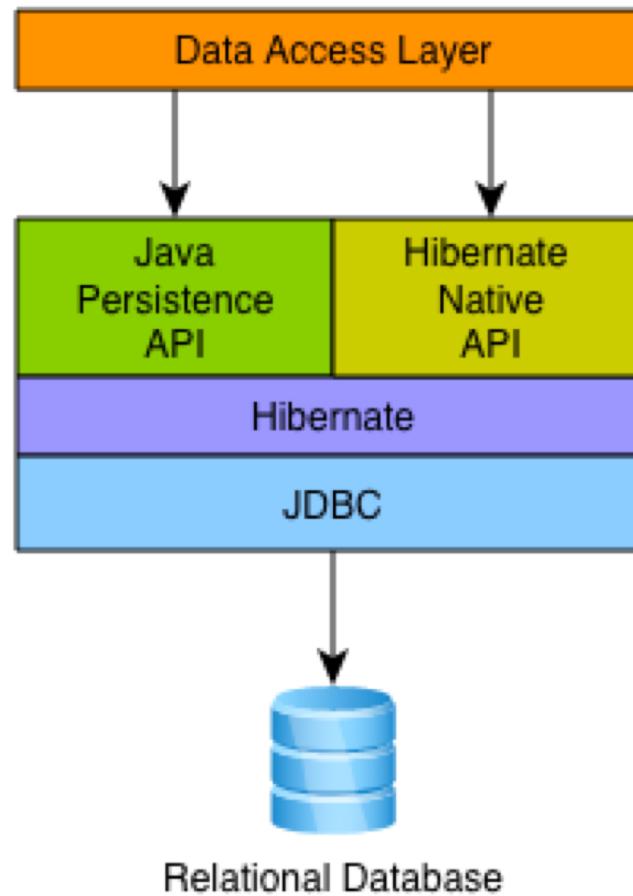
- Primer acercamiento
  - Clase = Tabla
  - Objeto = Tupla
  - Atributo del objeto = Columna de la tabla



# Visión de alto nivel de la Arquitectura



## Visión de alto nivel de hibernate



## Clases principales de Hibernate

- SessionFactory

- Es una caché (inmutable) de mappings compilados para una base de datos.
- Es un factory para las sesiones .

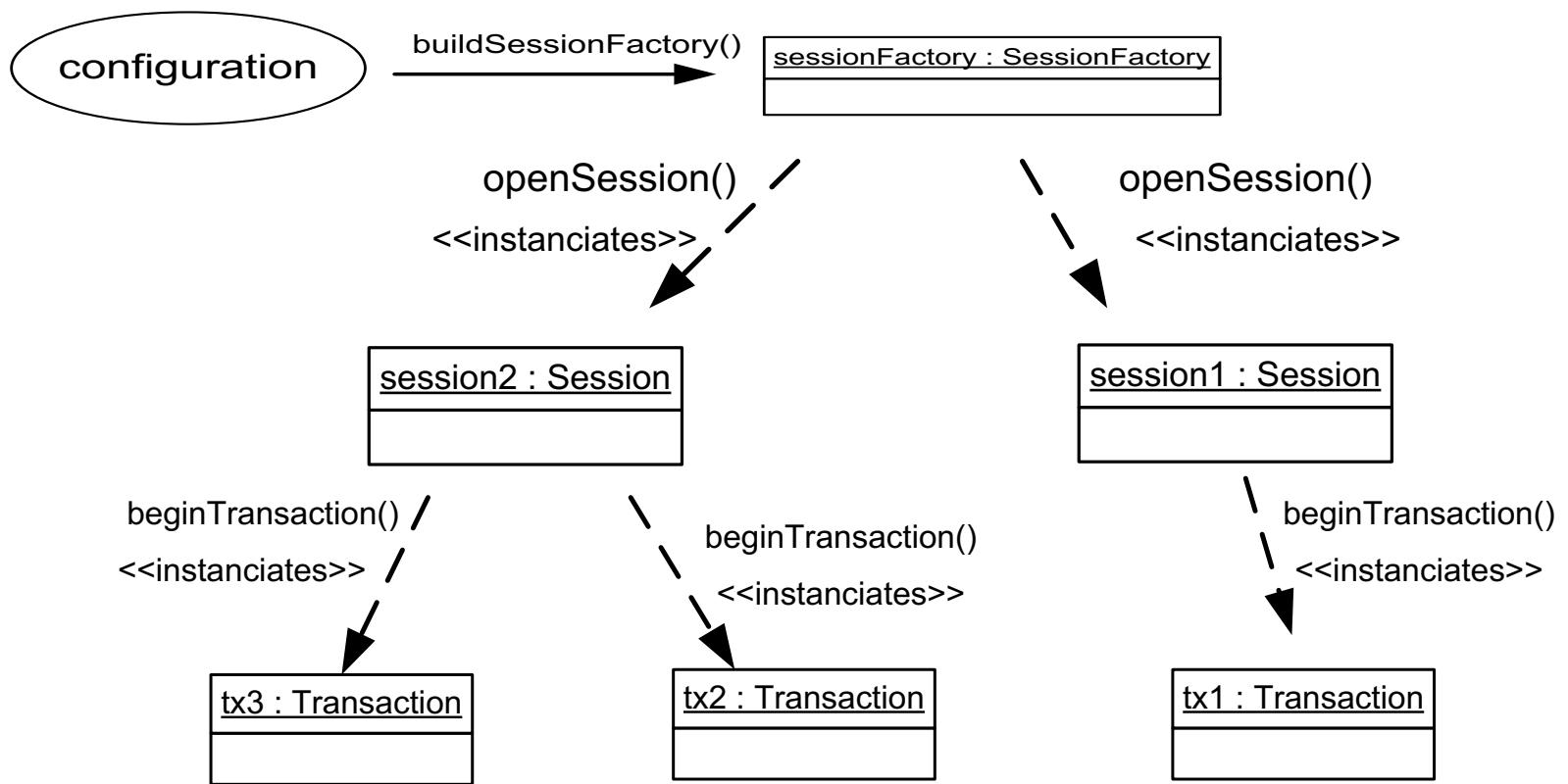
- Session

- Representa una “conversación” entre la aplicación y el almacenamiento persistente.
- Es un factory para las transacciones .
- Mantiene una caché de objetos persistentes, usada cuando se navega el grafo de objetos o para buscar objetos por identificador.

- Transaction

- Usadas por la aplicación para especificar unidades de trabajo atómicas.

# Arquitectura - Relaciones



## Configuración de Hibernate

- ¿Qué se debe configurar en Hibernate?
  - Cómo obtener una conexión JDBC.
  - Dialecto de la base datos.
  - Manejo de transacciones.
  - Mappings de las clases persistentes.
- ¿Cómo se configura?
  - A través de un descriptor, el mismo puede ser:
    - Un archivo XML.
    - Un archivo de Propiedades.
    - Programáticamente, es decir, por código Java.

# Configuración de Hibernate a través de un XML

- Archivo XML

- El nombre por defecto es hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
        <property
            name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver
        </property>
        <property name="hibernate.connection.url">jdbc:hsqldb:data/test
        </property>
        <property name="hibernate.connection.username">usuario</property>
        <property name="hibernate.connection.password">secreto</property>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <property name="show_sql">true</property>
        ...
        <!-- Mappings -->
        <mapping resource="paquete/miClase.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

- Aquí vemos para cada propiedad cual es su valor correspondiente.
- Inicialización desde Java:

```
Configuration cfg = new Configuration().configure();
SessionFactory sf = cfg.buildSessionFactory();
```

## Clases Persistentes

- Las clases persistentes son aquellas clases de las aplicaciones, que representan las entidades del negocio y cuyas instancias se pretenden almacenar.
- Para que Hibernate funcione correctamente, es recomendable que estas clases sigan un conjunto simple de reglas, conocidas como modelo de programación POJO (Plain Old Java Object).

## Plain Old Java Object (POJO) & Hibernate

- Hay cuatro reglas principales a seguir:
  - Implementar un constructor sin argumentos.
  - Proveer un identificador (opcional).
  - Son preferibles las clases no-finales (opcional).
  - Declarar setters y getters para los campos persistentes (opcional).

## Plain Old Java Object (POJO) & Hibernate

- Implementar constructor sin argumentos
- Las clases persistentes deben tener el constructor por defecto (el cual puede ser no-público) para que Hibernate pueda instanciar los objetos usando `Constructor.newInstance()`.
- Se recomienda tener el constructor por defecto con al menos visibilidad “package”, para la generación en tiempo de ejecución de proxies en Hibernate

## Plain Old Java Object (POJO) & Hibernate

- Proveer un identificador (opcional)
  - Esta propiedad se mapea a la columna de clave primaria de la tabla de la base de datos.
  - El identificador es opcional. Se puede omitir y dejar que Hibernate controle internamente los identificadores de los objetos. Sin embargo, no se recomienda esta práctica.
  - Algunas funcionalidades estarán disponibles solamente para las clases que declaran un identificador.
  - Se recomienda declarar identificadores con nombres significativos en las clases persistentes. También se recomienda usar tipos no primitivos (que puedan ser *null*).

## Plain Old Java Object (POJO) & Hibernate

- Son preferibles las clases no-finales (opcional)
  - Una característica central de Hibernate son los proxies, y para poder utilizarlos es necesario que las clases persistentes sean *no-final* o que implementen una *interface* con todos los métodos públicos.
  - Se pueden persistir con Hibernate clases finales que no implementen una *interface*, pero entonces no se podrán usar proxies para las asociaciones lazy, lo cual limita las opciones a la hora de hacer ajustes de performance.
  - Se debe evitar también declarar métodos *public final* en las clases *no-finales*. Si se desea usar métodos *public final*, se deben explícitamente deshabilitar los proxies seteando `lazy="false"`.

## Plain Old Java Object (POJO) & Hibernate

- Declarar setters y getters para los campos persistentes (opcional)
  - Es mejor tener una indirección entre el esquema relacional y las estructuras internas de datos de las clases.
  - Por defecto, Hibernate persiste propiedades del estilo *Javabeans*, y reconoce métodos de la forma `getFoo()`, `isFoo()` y `setFoo(Foo foo)`.
  - Se puede utilizar el acceso directo a las propiedades de la clase, en caso de ser necesario.
  - No es necesario que las propiedades sean declaradas públicas. Hibernate puede persistir propiedades con getters y setters con visibilidad `default`, `protected` o `private`.

## Mappings en Hibernate

- El mapeo objeto/relacional se define en un archivo XML o mediante el uso de annotations.
- El lenguaje de mapeo esta centrado en la declaración de las clases persistentes, y no en la declaración de las tablas.
- Se pueden escribir los archivos XML de mapeo a mano o con alguna de las herramientas existentes, como ser XDoclet y otras.
- Es una buena práctica mapear solamente una clase persistente (o una sola jerarquía de clases persistentes) en un archivo de mapeo y llamar a éste como la superclase, por ejemplo: Cat.hbm.xml, Dog.hbm.xml, o si se utiliza herencia, Animal.hbm.xml.

# Mappings en Hibernate

- La estructura básica es:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-
    //Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate
-mapping-3.0.dtd">

<hibernate-mapping>
[ ... ]
</hibernate-mapping>
```

# Ejemplo Mapping

```
package prueba;
public class Persona {
    private Long idPersona;
    private String nombre;
    private Date fechaNacimiento;
    public Date getFechaNacimiento() {
        return fechaNacimiento;
    }
    public void setFechaNacimiento(Date fechaNacimiento) {
        this.fechaNacimiento =
            fechaNacimiento;
    }
    public Long getIdPersona() {
        return idPersona;
    }
    public void setIdPersona(Long idPersona) {
        this.idPersona = idPersona;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

## Persona.hbm.xml (Archivo Mapping)

```
<hibernate-mapping package="prueba">
    <class name="Persona" table="PERSONA">
        <id name="IdPersona" column="ID_PERSONA">
            <generator class="native"/>
        </id>
        <property name="nombre" not-null="true"/>
        <property name="fechaNacimiento"/>
    </class>
</hibernate-mapping>
```

## Mappings en Hibernate

- Uso de annotations:

```
@Entity(name = "Contact")
public static class Contact {

    @Id
    private Integer id;

    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    //Getters and setters are omitted for brevity
}
```

## Tag <class>

- Se utiliza para mapear una clase persistente

```
<class name="ClassName"
      table="tableName"
      discriminator-value="discriminator_value"
      mutable="true|false"
      dynamic-update="true|false"
      dynamic-insert="true|false"
      select-before-update="true|false"
      where="arbitrary sql where
condition"           lazy="true|false"
      rowid="rowid"
      abstract="true|false"
      ...
/>
```

## Tag <class> - atributos

name:

- El nombre completo de la clase.

table (opcional - default → el nombre de la clase):

- El nombre de la tabla en la base.

discriminator-value (opcional - default → nombre de la clase):

- Un valor que distingue clases individuales, se usa para comportamiento polimórfico.

mutable (opcional, default → true):

- Especifica si las instancias son mutables o no.

dynamic-update (optional, default → false):

- Especifica si la sentencia SQL UPDATE sólo debe contener los campos que cambiaron.

dynamic-insert (optional, default → false):

- Especifica si la sentencia INSERT debe ser generada sólo con las columnas que son not-null .

## Identificadores Simples - Elemento id

- Todas las clases persistentes deben declarar la columna de la clave primaria.
- Tendrán una variable que actuará como identificador.

```
<id      name="propertyName"
          type="typename"
          column="column_name"
          unsaved-value="null|any|none|undefined|id_value"
          access="field|property|ClassName">
          node="element-name|@attribute-
          name|element/@attribute| . "
          <generator class="generatorClass"/>
</id>
```

name:

- El nombre de variable de instancia que se utiliza como identificador.

type (opcional):

- Un nombre que indica el tipo de hibernate a usar.

## Identificadores Simples - Elemento id

column (opcional - default → nombre de la variable):

- El nombre de la columna de la clave primaria.

unsaved-value (opcional - default → valor por defecto de Java):

- Un valor de la variable de instancia que indica que una instancia es nueva (no ha sido guardada); se utiliza para distinguir entre una instancia desacoplada (detached) que fueron guardadas o cargadas por sesiones anteriores.

access (opcional - defaults → property):

- La estrategia a utilizar para acceder el valor de la propiedad.

## Elemento <property>

- Utilizando el tag <property> se mapean las propiedades estilo JavaBean de las clases.

```
<property name="propertyName"
          column="column_name"
          type="typename"
          update="true|false" insert="true|false"
          formula="arbitrary SQL expression"
          access="field|property|ClassName"
          lazy="true|false"
          unique="true|false"
          not-null="true|false"
          generated="never|insert|always"
          ...
/>
```

## Atributos - Elemento <property>

name:

- El nombre de la propiedad.

column (opcional - default → nombre de la propiedad):

- El nombre de la columna a la que mapeará la propiedad.

type (opcional):

- El nombre que indica el tipo a usar (básicos o custom)

formula (opcional):

- Una expresión SQL que define el valor para una propiedad calculada. Dichas propiedades no poseen una columna

## Atributos - Elemento <property>

access (opcional - default → property):

- La estrategia que debe usar hibernate para acceder a la propiedad.

lazy (opcional - default → false):

- Especifica si la propiedad debe ser cargada lazy cuando la variable de instancia es accedida por primera vez .

unique (opcional):

- Permite la generación de un restricción unique.

not-null (optional):

- Permite la generación de una restricción not-null para la columna.

generated (opcional - default → never):

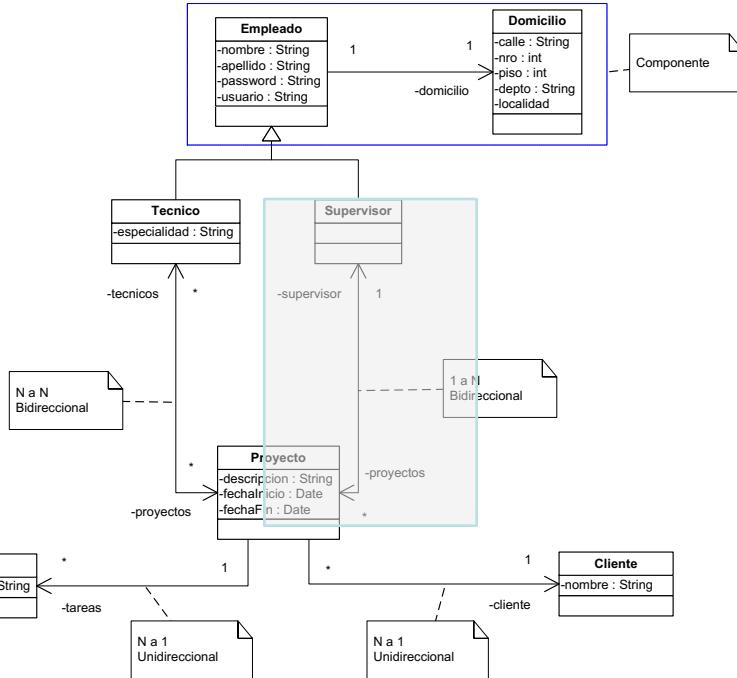
- Especifica si el valor de la propiedad es generada por la base de datos.

## Colecciones

- Hibernate permite mapear distintos tipos de colecciones:
  - Colecciones indexadas
    - List → <list>
    - Map → <map>
  - Colecciones no indexadas:
    - Set → <set>: no permite repetidos
    - List → <bag> (una lista en la que el orden no se mantiene)  
→ <idbag>
  - Colecciones Ordenadas (Sorted)
    - SortedSet → <set order="orden"/>
    - SortedMap → <map order="orden"/>

## Asociaciones - Uno a muchos <one-to-many>

- Relaciona las tablas de dos clases a través de una clave foránea sin la intervención de una tabla intermedia.
- Se utiliza dentro de una colección.



<one-to-many

```
class="ClassName"
not-found="ignore|exception"
entity-name="EntityName"
node="element-name"
embed-xml="true|false"
/>>
```

<set name="projects"

```
inverse="true"
cascade="all">
<key column="oid_emp" />
<one-to-many class="Project" />
```

</set>

## <one-to-many>

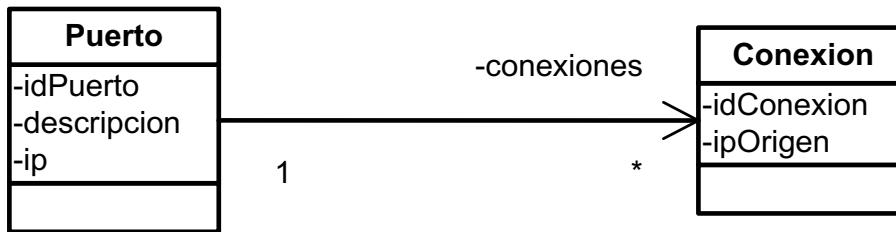
class (requerido):

- El nombre de la clase asociada

not-found (opcional - default → exception):

- Especifica cómo se manejan los identificadores que referencian a filas inexistentes; ignore tratará las filas inexistentes como asociaciones nulas.
- Se lleva la clave primaria de la clase que contiene el <one-to-many> a la clase indicada en class.
- <one-to-many> no necesita declarar ninguna columna, tampoco es necesario especificar ninguna tabla.

# Unidireccionales: Uno a Muchos (Mapping)



```

<hibernate-mapping
  package="com.lifia.hibernate.model.uni">

<class name="Puerto">

  <id name="idPuerto" column="ID_PUERTO">
    <generator class="native"/>
  </id>

  <set name="conexiones" cascade="all">
    <key column="ID_PUERTO"
        not-null="true"/>
    <one-to-many class="Conexion"/>
  </set>

  <property name="descripcion" />
  <property name="ip" />

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

```

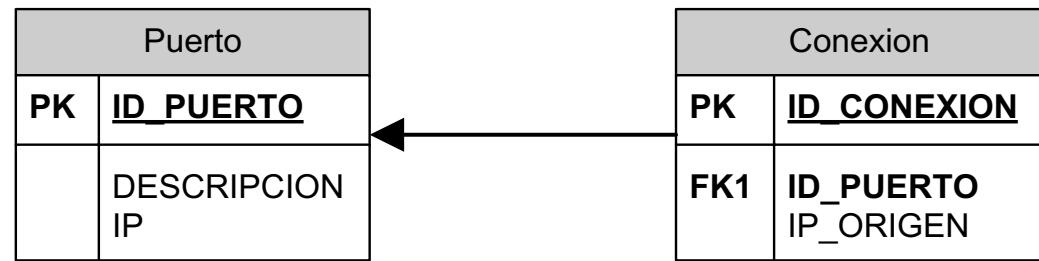
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">

<class name="Conexion">

  <id name="idConexion" column="ID_CONEXION">
    <generator class="native"/>
  </id>

  <property name="ipOrigen" />

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



## Asociaciones - Muchos a uno <many-to-one>

- Este elemento permite asociar dos clases persistentes.
- Crea una referencia de integridad entre dos tablas.

```
<many-to-one  
    name="propertyName"  
    column="column_name"  
    class="ClassName"  
    cascade="cascade_style"  
    property-ref="propertyNameFromAssociatedClass"  
    unique="true|false"  
    not-null="true|false"  
    lazy="proxy|no-proxy|false"  
    not-found="ignore|exception" ... />
```

name (obligatorio):

- El nombre de la propiedad .

column:

- El nombre de la columna de la clave foránea.

## Asociaciones - Muchos a uno <many-to-one> (cont.)

class (default → el tipo de la propiedad determinado por reflexión):

- El nombre de la clase asociada.

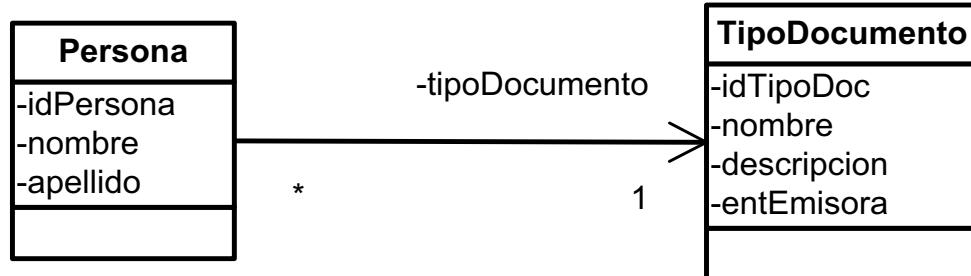
cascade:

- Especifica cuales operaciones deben ser propagadas desde el objeto padre al objeto asociado.

unique:

- Si se especifica la relación de transforma en uno a uno (por defecto tiene valor false).
- Se puede usar o no en una colección.

## Unidireccionales: Muchos a Uno (Mapping)



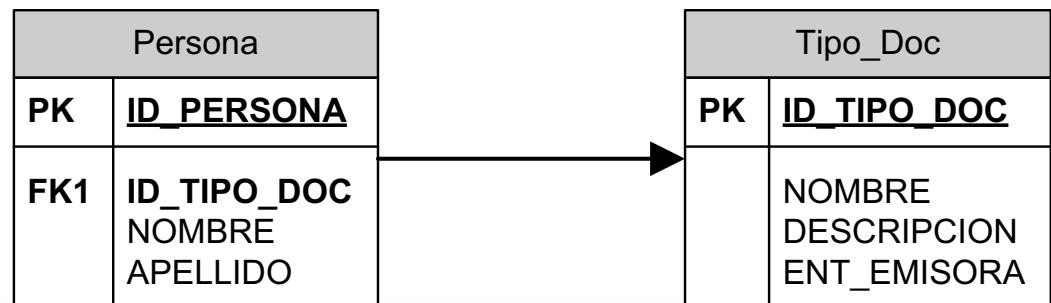
```

<hibernate-mapping package="model.uni">

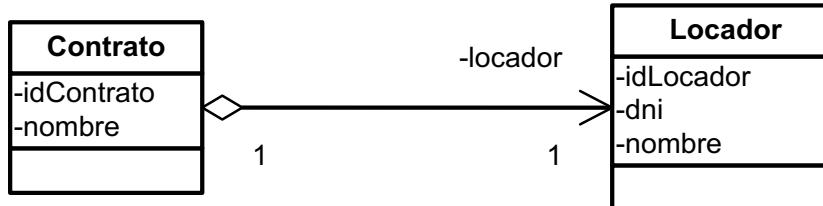
<class name="Persona">
  <id name="idPersona" column="ID_PERSONA">
    <generator class="native"/>
  </id>
  <many-to-one name="tipoDocumento"
    column="ID_TIPO_DOC"
    not-null="true"
    cascade="all"/>
  <property name="nombre"/>
  <property name="apellido"/>
</class>
</hibernate-mapping>
  
```

```

<hibernate-mapping package="model.uni">
<class name="TipoDocumento"
  table="Tipo_Doc">
  <id name="idTipoDoc"
    column="ID_TIPO_DOC">
    <generator class="native"/>
  </id>
  <property name="nombre" />
  <property name="descripcion" />
  <property name="entEmisora"
    column="ENT_EMISORA"/>
</class>
</hibernate-mapping>
  
```



## Unidireccionales: Uno a Uno (Mapping)



```
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">
<class name="Contrato" >
  <id name="idContrato" column="ID_CONTRATO">
    <generator class="native"/>
  </id>
  <many-to-one name="locador"
    column="ID_LOCADOR"
    unique="true"
    not-null="true"
    cascade="all" />
  <property name="nombre" column="NOMBRE"/>
</class>
</hibernate-mapping>
```

```
<hibernate-mapping
  package="com.lifia.hibernate.model.uni">
<class name="Locador">
  <id name="idLocador" column="ID_LOCADOR">
    <generator class="native"/>
  </id>
  <property name="dni" column="DNI" />
  <property name="nombre" column="NOMBRE"/>
</class>
</hibernate-mapping>
```

Contrato		Locador	
PK	<u>ID CONTRATO</u>	PK	<u>ID LOCADOR</u>
FK1,U1	ID LOCADOR NOMBRE		DNI NOMBRE

## Asociaciones - Muchos a Muchos <many-to-many>

- Se mapean con el elemento <many-to-many>.
- Llevan una tabla dedicada.
- Se utiliza dentro de una colección

```
<many-to-many column="column_name"  
               class="ClassName"  
               unique="true | false"  
               ... />
```

class (obligatorio):

- El nombre de la clase asociada.

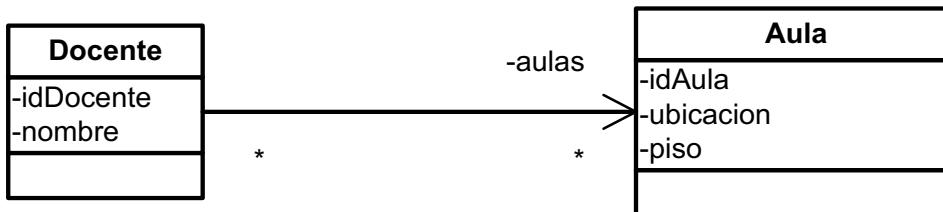
column:

- El nombre de la columna de la clave foránea del elemento.

unique:

- Si se pone en “true” la asociación se transforma en uno a muchos.

## Unidireccionales: Muchos a Muchos (Mapping)



```

<hibernate-mapping>
<class name="Docente" >

    <id name="idDocente"
        column="ID_DOCENTE" >
        <generator class="native"/>
    </id>

    <set name="aulas"
        table="DOCENTE_AULA"
        cascade="all">
        <key column="ID_DOCENTE" />
        <many-to-many column="ID_AULA"
            class="Aula" />
    </set>

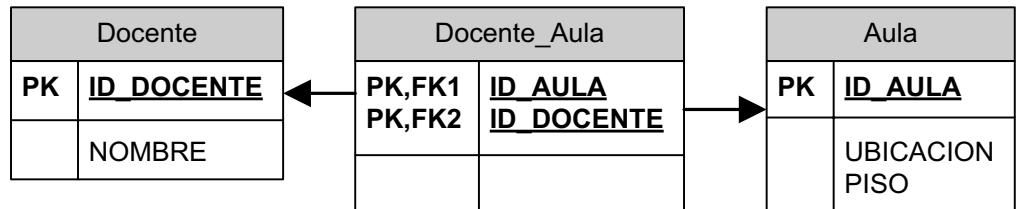
    <property name="nombre" />
</class>
</hibernate-mapping>
  
```

```

<hibernate-mapping>
<class name="Aula" >

    <id name="idAula" column="ID_AULA" >
        <generator class="native"/>
    </id>

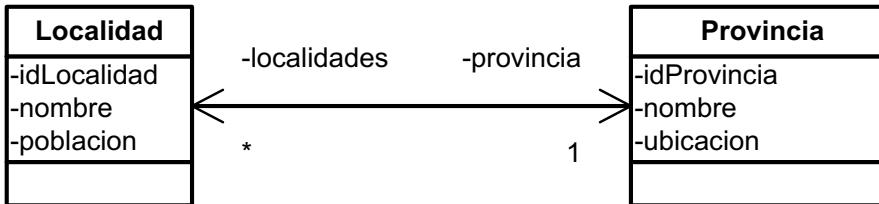
    <property name="ubicacion" />
    <property name="piso" />
</class>
</hibernate-mapping>
  
```



## Asociaciones Bidireccionales

- Una *asociación bidireccional* permite la navegación desde ambos lados de la asociación. Se soportan dos tipos:
  - uno a muchos (one-to-many)
    - Conjuntos de valores (set o bag) en un lado y un valor simple en el otro.
  - Muchos a muchos (many-to-many)
    - Conjuntos de valores (set o bag) en ambos lados.
- ¿Cómo se declara?
  - Se especifica mapeando dos asociaciones a la misma tabla de la base y declarando una como `inverse`.

## Bidireccionales: Muchos a uno (Mapping)



```

<hibernate-mapping package="model.bi">
<class name="Localidad" >

    <id name="idLocalidad"
        column="ID_LOCALIDAD">
        <generator class="native"/>
    </id>

    <many-to-one name="provincia"
        column="ID_PROVINCIA"
        not-null="true"
        cascade="all"/>

    <property name="nombre" />
    <property name="poblacion" />

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

Localidad		Provincia	
PK	ID_LOCALIDAD	PK	ID_PROVINCIA
FK1	ID_PROVINCIA NOMBRE POBLACION		NOMBRE UBICACION

```

<hibernate-mapping package="model.bi">
<class name="Provincia" >

    <id name="idProvincia"
        column="ID_PROVINCIA">
        <generator class="native"/>
    </id>

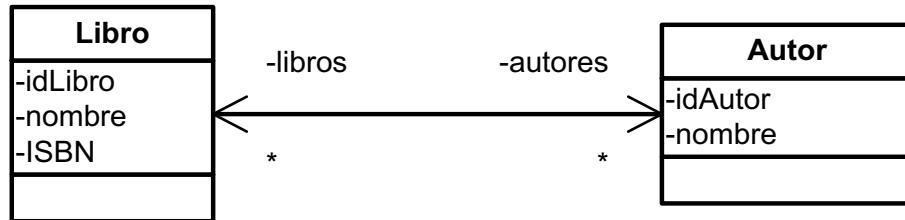
    <set name="localidades"
        inverse="true" cascade="all">

        <key column="ID_PROVINCIA"/>
        <one-to-many class="Localidad"/>
    </set>

    <property name="nombre" />
    <property name="ubicacion" />

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

## Bidireccionales: Muchos a Muchos (Mapping)



```

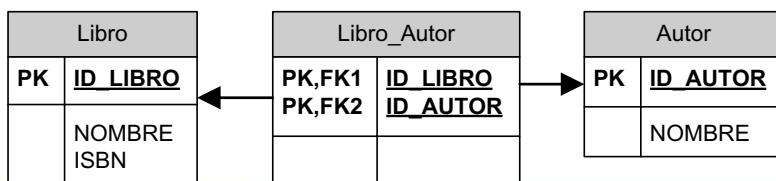
<hibernate-mapping package="model">
<class name="Libro">

    <id name="idLibro" column="ID_LIBRO">
        <generator class="native"/>
    </id>
    <set name="autores">
        table="LIBRO_AUTOR"
        cascade="all">
            <key column="ID_LIBRO" />
            <many-to-many column="ID_AUTOR"
                class="Autor"/>
    </set>
</class>
</hibernate-mapping>
  
```

```

<hibernate-mapping package="model">
<class name="Autor">

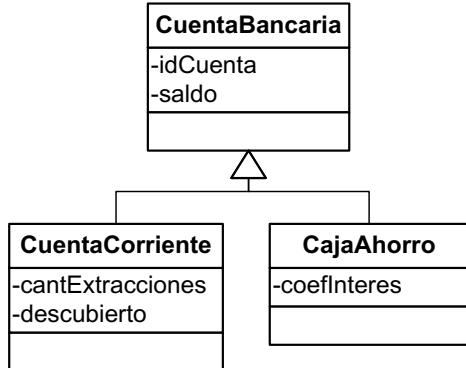
    <id name="idAutor" column="ID_AUTOR">
        <generator class="native"/>
    </id>
    <set name="libros">
        table="LIBRO_AUTOR"
        inverse="true" >
            <key column="ID_AUTOR" />
            <many-to-many column="ID_LIBRO"
                class="Libro"/>
    </set>
</class>
</hibernate-mapping>
  
```



## Herencia - Jerarquías - Mappings

- Hibernate soporta tres estrategias para mapear Jerarquías
  - Tabla por jerarquía.
  - Tabla por clase.
  - Tabla por clase concreta.

# Tabla por Jerarquía

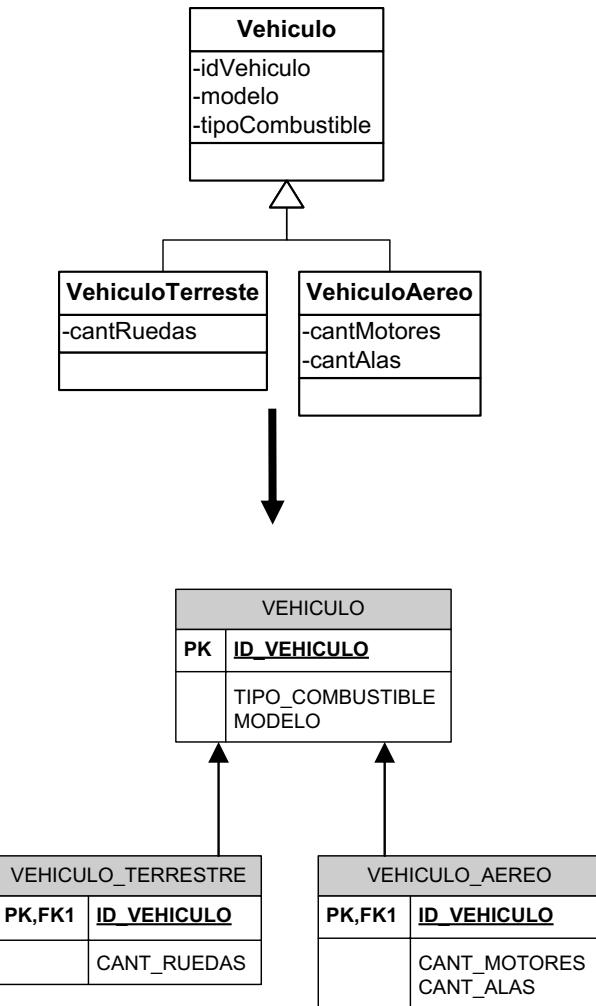


CUENTA_BANCARIA	
PK	ID CUENTA
	TIPO CUENTA SALDO CANT_EXTRACCIONES COEF_INTERES

```
<hibernate-mapping
  package="com.lifia.hibernate.model.hierarchy">

  <class      name="CuentaBancaria"
  table="CUENTA_BANCARIA">
    <id name="idCuenta" column="ID CUENTA">
      <generator class="native"/>
    </id>
    <discriminator column="TIPO CUENTA"
      type="string"/>
    <property name="saldo" column="SALDO"/>
    <subclass name="CuentaCorriente"
      discriminator-value="CC">
      <property name="cantExtracciones"
      column="CANT_EXTRACCIONES"/>
      <property name="descubierto"
      column="DESCUBIERTO"/>
    </subclass>
    <subclass name="CajaAhorro"
      discriminator-value="CA">
      <property name="coefInteres"
      column="COEF_INTERES"/>
    </subclass>
  </class>
</hibernate-mapping>
```

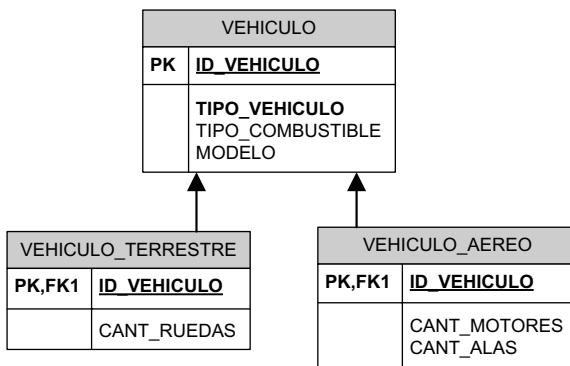
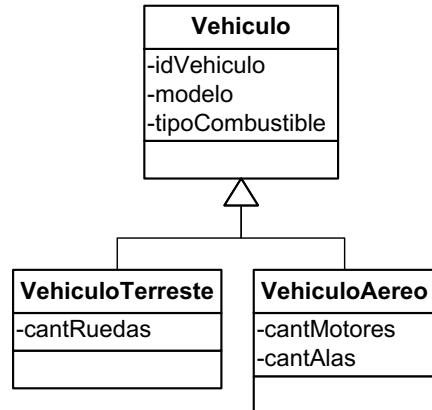
# Tabla por clase



```

<hibernate-mapping
  package="com.lifia.hibernate.model.hierarchy">
  <class name="Vehiculo" table="VEHICULO" >
    <id name="idVehiculo" column="ID_VEHICULO">
      <generator class="native"/>
    </id>
    <property name="modelo"/>
    <property name="tipoCombustible"
      column="TIPO_COMBOUSTIBLE"/>
  <joined-subclass name="VehiculoTerreste"
    table="VEHICULO_TERRESTRE">
    <key column="ID_VEHICULO"/>
    <property name="cantRuedas"
      column="CANT_RUEDAS"/>
  </joined-subclass>
  <joined-subclass name="VehiculoAereo"
    table="VEHICULO_AEREO">
    <key column="ID_VEHICULO"/>
    <property name="cantMotores"
      column="CANT_MOTORES"/>
    <property name="cantAlas"
      column="CANT_ALAS"/>
  </joined-subclass>
  </class>
</hibernate-mapping>
  
```

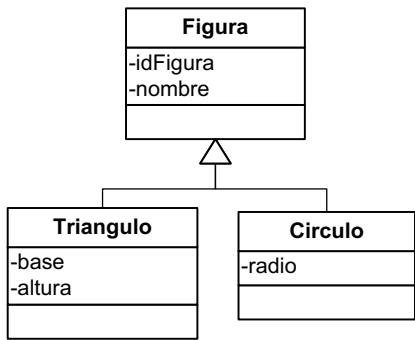
# Tabla por clase con discriminante



```

<hibernate-mapping
  package="com.lifia.hibernate.model.hierarchy">
  <class name="Vehiculo" table="VEHICULO" >
    <id name="idVehiculo" column="ID_VEHICULO">
      <generator class="native"/>
    </id>
    <discriminator column="TIPO_VEHICULO" type="string"/>
    <property name="modelo"/>
    <property name="tipoCombustible"
      column="TIPO_COMBOUSTIBLE"/>
    <subclass name="VehiculoTerrestre"
      discriminator-value="VT">
      <join table="VEHICULO_TERRESTRE">
        <key column="ID_VEHICULO"/>
        <property name="cantRuedas"
          column="CANT_RUEDAS"/>
      </join>
    </subclass>
    <subclass name="VehiculoAereo"
      discriminator-value="VA">
      <join table="VEHICULO_AEREO">
        <key column="ID_VEHICULO"/>
        <property name="cantMotores"
          column="CANT_MOTORES"/>
        <property name="cantAlas" column="CANT_ALAS"/>
      </join>
    </subclass>
  </class>
</hibernate-mapping>
  
```

# Tabla por Clase Concreta



```
<hibernate-mapping
  package="com.lifia.hibernate.model.hierarchy">

  <class name="Figura" table="FIGURA">
    <id name="idFigura" column="ID FIGURA">
      <generator class="hilo"/>
    </id>
    <property name="nombre"/>
    <union-subclass name="Circulo" table="FIGURA_CIRCULO">
      <property name="radio"/>
    </union-subclass>
    <union-subclass name="Triangulo" table="FIGURA_TRIANGULO">
      <property name="altura"/>
      <property name="base"/>
    </union-subclass>
  </class>
</hibernate-mapping>
```

FIGURA_CIRCULO	
PK,FK1	ID FIGURA
	NOMBRE RADIO

FIGURA_TRIANGULO	
PK,FK1	ID FIGURA
	NOMBRE ALTURA BASE

## Control optimista de concurrencia

- Cuando se tienen ambientes de alta concurrencia y escalabilidad, se utiliza control de concurrencia optimista y versionamiento.
- Chequeo de versiones para detectar conflictos de actualización, se usa:
  - Números de versión.
  - Timestamps.
- Hibernate posee tres acercamientos:
  - Control de versiones por aplicación.
  - Sesión extendida y versionamiento automático.
  - Objetos desacoplados y versionamiento automático.

## Versionamiento

- El elemento `<version>` es opcional e indica que la tabla contiene datos versionados.
- Es útil si se planea usar transacciones largas.

```
<version column="version_column"
         name="propertyName"
         type="typename"
         access="field|property|ClassName"
         unsaved-value="null|negative|undefined"
         generated="never|always"
         insert="true|false"
      />
```

column (opcional - default → valor de la propiedad):

- El nombre de la columna que contiene el número de versión.

name:

- El nombre de la propiedad persistente.

type (optional - default → integer):

- El tipo del número de versión.

## Recuperar un objeto

Métodos para Recuperar:

- `session.load()`
  - Recupera el objeto con un identificador, sino levanta una excepción

```
Persona persona = (Persona)  
session.load(Persona.Class, identificador);
```

- `session.get()`
  - Recupera el objeto con un identificador, si no existe retorna null

```
Persona persona = (Persona)  
session.get(Persona.Class, identificador);
```