IES CHAN DO MONTE
C.S. de Desenvolveento de Aplicacions Multiplataforma

UNIDAD 4: HIBERNATE. Mapeo con anotacións

Índice

1. Mapeo con anotacións

1.1 INTRODUCCIÓNS

Dende a versión 3, Hibernate soporta o uso de anotacións de Java 5 para especificar a relación entre as clases e as táboas da base de datos.

Con Hibernate podemos facer que as nosas clases se almacenen nunha base de datos facendo ficheiros de mapeo XML, nos que se indica como se garda en base de datos cada clase. **Coas anotacións de Hibernate, podemos facer isto mesmo, pero reemprazando os ficheiros XML poñendo anotacións Java na mesma clase**.

Vantaxes das Anotacións:

- As anotacións permiten **reducir a cantidade de ficheiros** que temos que manter, xa que só é necesario o .java. Ademais, escribiremos menos, xa que a maioría das opcións teñen valores por defecto. Isto significa que na maioría dos casos podemos aforrar as anotacións, bastaría con especificar que a clase é unha entidade, cal é o campo identificador e pouco máis.
- Compatibilidade con JPA: Unha das principais vantaxes das anotacións de Hibernate é que son **compatibles coas anotacións de JPA** (Java Persistente API, a capa de persistencia de EJB 3.0). Isto permite unha maior flexibilidade e interoperabilidade entre diferentes plataformas e tecnoloxías.

Unha **diferencia importante entre usar o ficheiro de mapeo hbm.xml e as anotacións** é que no ficheiro é obrigatorio indicar todas as propiedades que queremos que persistan en base de datos, mentres que usando as anotacións iso non é necesario. Usando anotacións persisten todas as propiedades que teñan os métodos get/set.

1.2 persistentes

Especificacións que deben cumprir as clases en Hibernate para traballar con anotacións:

- **Clases POJO**: Hibernate traballa con clases que son POJOs (Plain Old Java Object - Obxectos Simples). Estas clases deben ser obxectos Java sinxelos que non requiren unha clase pai, nin implementan interfaces específicas1.
- **Constructor sen argumentos**: As clases deben implementar un constructor sen argumentos. Hibernate require isto para poder crear instancias da entidade.
- **Propiedade identificadora:** Cada entidade debe proporcionar unha propiedade identificadora, que actúa como a clave primaria. Esta propiedade debe estar anotada con @Id.
- **Métodos getter e setter:** Para cada atributo da clase, deben existir métodos getter e setter correspondentes. Hibernate utiliza estes métodos para acceder aos campos.
- **Implementación da interface Serializable**: Aínda que non é obrigatorio, é recomendable que as clases implementen a interface Serializable. Isto facilita a interoperabilidade e a persistencia dos datos.
- **Uso de anotacións**: As clases deben usar anotacións para indicar a Hibernate como mapear os atributos da clase ás columnas da base de datos. As anotacións como @Entity, @Table, @Column, e outras, son utilizadas para este propósito.

Dada a seguinte base de datos:

```
create database Colegio
go
USE Colegio;
GO
CREATE TABLE Alumnos (
    idMatricula CHAR(6),
    DNI CHAR(9) NOT NULL,
    Nome VARCHAR(100) NOT NULL,
    Apelido1 VARCHAR(100) NOT NULL,
    Apelido2 VARCHAR(100) NULL ,
    DataNacemento DATE NOT NULL,
    CONSTRAINT pk_alumnos PRIMARY KEY (idMatricula),
    CONSTRAINT uk_dni UNIQUE (DNI),
    CONSTRAINT ck_dni CHECK (DNI LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][A-Z]')
);
```

Utilizando ficheiro Alumno.hbm.xml

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

<hibernate-mapping>
    <class name="POJOS.Alumno" table="Alumnos">
        <id name="idMatricula" column="idMatricula">
            <generator class="assigned"/>
        </id>
        <property name="dni" column="DNI" not-null="true" unique="true"/>
        <property name="nome" column="Nome" not-null="true"/>
        <property name="apelido1" column="Apelido1" not-null="true"/>
        <property name="apelido2" column="Apelido2"/>
        <property name="dataNacemento" column="DataNacemento" type="date" not-null="true"/>
    </class>
</hibernate-mapping>
```

O POJO coas anotacións que mapean a tabla Alumno: :

```
@Entity
@Table(name = "Alumnos", uniqueConstraints = {
    @UniqueConstraint(name = "uk_dni", columnNames = "DNI")
})
public class Alumno {

    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    @Column(name = "DNI", nullable = false)
    private String dni;

    @Column(name = "Nome", nullable = false)
    private String nome;

    @Column(name = "Apelido1", nullable = false)
    private String apelido1;

    @Column(name = "Apelido2")
    private String apelido2;

    @Column(name = "DataNacemento", nullable = false)
    @Temporal(TemporalType.DATE)
    private Date dataNacemento;

    // Getters e setters
}
```

Explicación das anotacións utilizadas no exemplo anterior:

- **`@Entity`:** Esta anotación indica que a clase `Alumno` é unha entidade, o que significa que representa unha táboa na base de datos.
- **`@Table(name = "Alumnos")`:** Esta anotación vincula a clase `Alumno` coa táboa `Alumnos` na base de datos.

  `@UniqueConstraint` define unha restrición de unicidade na columna DNI a nivel de tabla. Isto significa que a base de datos non permitirá inserir dous alumnos co mesmo DNI. A propiedade name especifica o nome da restrición, e columnNames especifica a(s) columna(s) á(s) que se aplica a restrición.

  Se non se especifican schema nin catalog na anotación @Table, Hibernate utilizará os valores predeterminados da conexión á base de datos.

```
@Entity
@Table(name = "Alumnos",
       catalog = "Colegio",
       schema = "dbo",
       uniqueConstraints = {
           @UniqueConstraint(name = "uk_dni", columnNames = "DNI")
       })
public class Alumno {
    // campos, getters e setters
```

- **`@Id`:** Esta anotación indica que o campo `idMatricula` é a chave primaria da táboa `Alumnos`.
  - Non se especificou ningunha estratexia de xeración de clave, o que significa que se debe proporcionar os valores para idMatricula manualmente ao crear novas entidades. Se se quisiera que a base de datos xerase automaticamente os valores, poderías usar a anotación **@GeneratedValue.**
  - Por exemplo, se estás a usar MySQL e queres que os valores se autoincrementen, poderías usar **@GeneratedValue(strategy = GenerationType.IDENTITY)**. Isto indicaría que estás a usar a estratexia de autoincremento do servidor da base de datos.
- **@Column(name = "idMatricula"):** Esta anotación vincula o campo idMatricula coa columna idMatricula na táboa Alumnos.
- **`@Column(name = "DNI", nullable = false)`:** Esta anotación vincula o campo `dni` coa columna `DNI` na táboa `Alumnos`. A propiedade `nullable = false` indica que esta columna non pode ter valores nulos.
- **`@Column(name = "Nome", nullable = false)`:** Esta anotación vincula o campo `nome` coa columna `Nome` na táboa `Alumnos`. A propiedade `nullable = false` indica que esta columna non pode ter valores nulos.
- **`@Column(name = "Apelido1", nullable = false)`:** Esta anotación vincula o campo `apelido1` coa columna `Apelido1` na táboa `Alumnos`. A propiedade `nullable = false` indica que esta columna non pode ter valores nulos.
- **`@Column(name = "Apelido2")`:** Esta anotación vincula o campo `apelido2` coa columna `Apelido2` na táboa `Alumnos`.
- **`@Column(name = "DataNacemento", nullable = false)`:** Esta anotación vincula o campo `dataNacemento` coa columna `DataNacemento` na táboa `Alumnos`. A propiedade `nullable = false` indica que esta columna non pode ter valores nulos.
- **`@Temporal(TemporalType.DATE)`:** Esta anotación indica que só a data será almacenada na base de datos, ignorando a hora e os segundos.

Todas as clases de entidades persistentes deben ter unha propiedade que sirva como identificador, se queremos usar o conxunto completo de funcionalidades que nos ofrece Hibernate.

1.3 Estrategias de generación de clave

Declaración columna de clave primaria: < id>

As clases mapeadas teñen que declarar a columna de clave primaria da táboa da base de datos.

```
<id name="id" type="int">
    <column name="id" />
    <generator class="assigned" />
</id>
```

Hibernate necesita coñecer cal é a *estratexia elixida para a xeración de claves primarias*. Moitas bases de datos usan claves primarias naturais, claves que teñen un significado no mundo real (CIF, NSS, ...). Nalgúns casos as claves naturais están compostas de varias columnas, o que fai o mantemento, consultas e a evolución do sistema máis difícil. Unha recomendación é utilizar claves artificiais (ou suplentes), que non teñen significado para a aplicación e que son xeradas automaticamente polo sistema, e definir restricións de unicidade nas claves naturais.

O elemento que define o mapeo da columna de clave primaria é a etiqueta `<id>`. Sintaxe:

```
<id
   name="nome do atributo"
   type="tipo de datos"
   column="nome  da columna"
   access="field|property|ClassName">
   <generator class="generatorClass"/>
</id>
```

- **name** (opcional): Nome do atributo da clase persistente que vai ser un identificador. Si se omite name, asúmese que a clase non ten propiedade identificadora.
- **type** (opcional): Nome que indica o tipo de datos de Hibernate.
- **column** (opcional - por defecto é o nome do atributo): Nome da columna de clave primaria na base de datos.
- **access** (opcional - property por defecto): Estratexia de acceso ao valor da propiedade.
- **generator** (opcional): Ao establecer unha clave primaria, débese especificar a forma en que esta se xera. Hibernate ten implementada varias formas de xerar a clave primaria, pero non todas as formas son portables a todos os SXBD. Todos os xeradores implementan a interface `org.hibernate.id.IdentifierGenerator`. Algúns xeradores incorporados son os seguintes:

  assigned: Esta estratexia permite á túa aplicación asignar o valor do identificador. Deixa á aplicación asignar un identificador ao obxecto antes de que se chame ao método save(). Esta é a estratexia por defecto se non se especifica un elemento <generator>.Isto é útil cando os teus identificadores teñen un significado no dominio do teu problema. Por exemplo, se estás mapeando unha clase Usuario e usas o nome de usuario como identificador, podes usar a estratexia assigned.

  increment: Esta estratexia xera identificadores incrementais. Cada vez que necesitas un novo identificador, Hibernate incrementa o valor máximo actual na columna do identificador. Esta estratexia é simple e eficiente, pero só debe usarse cando a túa aplicación ten acceso exclusivo á base de datos.

identity: Esta estratexia usa columnas de identidade proporcionadas pola base de datos. Moitas bases de datos soportan columnas de identidade que xeran automaticamente un valor único cada vez que se inserta unha nova fila. Esta estratexia é moi eficiente e permite á base de datos encargarse da xeración do identificador.

sequence: Esta estratexia usa secuencias de base de datos para xerar identificadores. Unha secuencia é un obxecto de base de datos que xera unha secuencia de números únicos. Cando necesitas un novo identificador, solicitas o seguinte valor da secuencia.

hilo e seqhilo: Estas estratexias usan un algoritmo "high/low" para xerar identificadores de forma eficiente. O algoritmo high/low xera identificadores que son únicos para a base de datos. Os valores high obtéñense dunha fonte global e fanse únicos engadindo un valor low. A estratexia seqhilo é similar, pero usa unha secuencia de base de datos para xerar os valores high.

uuid e guid: Estas estratexias xeran identificadores únicos a nivel global. uuid usa un algoritmo UUID para xerar un identificador de 128 bits, mentres que guid usa unha cadea GUID xerada pola base de datos.

native: Esta estratexia permite a Hibernate escoller a mellor estratexia baseándose nas capacidades da base de datos.

select e foreign: Estas estratexias son un pouco diferentes. select recupera un identificador asignado por un disparador da base de datos, mentres que foreign usa o identificador dun obxecto asociado.

Equivalencia en anotacións para a xeración de claves

- **assigned:** Esta estratexia permite á túa aplicación asignar o valor do identificador. Para isto, simplemente non usas a anotación **@GeneratedValue**. Por exemplo:

```
@Entity
public class Usuario {
    @Id
    private Long id;

    // máis campos, getters e setters aquí
}
```

Neste exemplo, a clase Usuario usa a estratexia assigned para o seu identificador. Antes de gardar un Usuario, debes asignar un valor ao campo id.

```
Usuario usuario = new Usuario();
usuario.setId(123L);  // asignas un valor ao identificador
session.save(usuario);  // gardas o usuario
```

- **increment:** Hibernate pode xerará automaticamente un valor incremental para o campo id cada vez que se cree unha nova instancia. @GeneratedValue usa un xerador chamado "increment". A anotación @GenericGenerator define este xerador "increment" para usar a estratexia "increment". Ejemplo:

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy
= "increment")
    private Long id;

    @Column(unique=true)
    private String numeroIdentificacion;

    // ...
}
```

- **identity**: Esta estratexia usa columnas de identidade proporcionadas pola base de datos. Para isto, usas a anotación `@GeneratedValue` con a estratexia `GenerationType.IDENTITY`. Por exemplo:

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // máis campos, getters e setters aquí
}
```

- **sequence:** Esta estratexia usa secuencias de base de datos para xerar identificadores. Para isto, usas a anotación `@GeneratedValue` con a estratexia `GenerationType.SEQUENCE`. Por exemplo:

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy
= "increment")
    private Long id;

    @Column(unique=true)
    private String numeroIdentificacion;

    // ...
}
```

  Neste exemplo, a anotación @GeneratedValue usa a estratexia GenerationType.SEQUENCE, que indica a Hibernate que debe usar unha secuencia de base de datos para xerar un valor único para o campo id cada vez que se cree unha nova instancia de MinhaClase.

  As secuencias son un recurso moi común en moitas bases de datos relacionais, e permiten xerar valores numéricos únicos de maneira eficiente. Cada vez que se solicita un novo valor da secuencia, a base de datos incrementa o valor da secuencia e devólveo, garantindo así que cada transacción obteña un valor único. Adisponibilidade e o comportamento das secuencias poden variar dependendo do sistema de xestión de bases de datos que estés a usar. Por exemplo, mentres que moitas bases de datos soportan secuencias, outras como MySQL non as soportan de maneira nativa. Neste último caso, Hibernate pode emular o comportamento das secuencias usando unha táboa especial.

- **hilo e seqhilo: Hibernate non soporta directamente estas estratexias con anotacións.**
- **uuid e guid**: Para xerar identificadores UUID, usas a anotación @GeneratedValue con a estratexia GenerationType.AUTO e un tipo de dato UUID. Por exemplo:

```
@Entity
public class MinhaClase {
    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(
        name = "UUID",
        strategy =
"org.hibernate.id.UUIDGenerator"
    )
    @Column(name = "id", updatable = false,
nullable = false)
    private String id;

    // ...
}
```

Un UUID (Universally Unique Identifier) é un identificador único universal que proporciona unha maneira de identificar de maneira única as entidades nun sistema distribuído. Os UUIDs son cadeas de caracteres de 36 caracteres de lonxitude que inclúen díxitos hexadecimais e guións, polo que ocupan máis espazo que os identificadores numéricos.

- **native:** Esta estratexia permite a Hibernate escoller a mellor estratexia baseándose nas capacidades da base de datos. Para isto, usas a anotación @GeneratedValue con a estratexia GenerationType.AUTO. Por exemplo:

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID id;

    // máis campos, getters e setters aquí
}
```

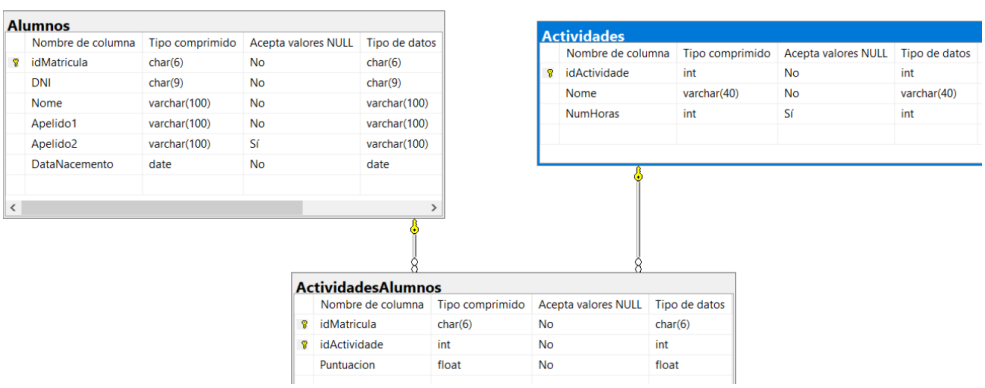
Claves compostas :

Define claves compostas por varios campos.

Utilizando os ficheiros de mapeo en Hibernate. Para unha clave composta, necesitamos definir unha clase de clave primaria que conteña os campos da clave composta.

Por exemplo: temos alumnos, actividades e queremos gardar as actividades que realizan los alumnos coas súas puntuacións.

Na base de datos temos:



Os POJOS serían:

```
public class Alumnos implements Serializable {
    private String idMatricula;
    private String dni;
    private String nome;
    private String apelido1;
    private String apelido2;
    private Date dataNacemento;

    // getters e setters
}
```

```
public class Actividades implements Serializable {
    private int idActividade;
    private String nome;
    private Integer numHoras;

    // getters e setters
}
```

A clase ActividadesAlumnosId que representa a clave composta:

```
public class ActividadesAlumnosId implements Serializable {
    private String idMatricula;
    private int idActividade;

    // getters e setters
}
```

E a clase ActividadesAlumnos:

```
public class ActividadesAlumnos implements Serializable {
    private ActividadesAlumnosId id;
    private float puntuacion;

    // getters e setters
}
```

Os FICHEIROS DE MAPEO serían:

Alumnos.hbm.xml

```
<class name="Alumnos" table="ALUMNOS">
    <id name="idMatricula" column="ID_MATRICULA" type="string">
        <generator class="assigned"/>
    </id>
    <property name="dni" column="DNI"/>
    <property name="nome" column="NOME"/>
    <property name="apelido1" column="APELIDO1"/>
    <property name="apelido2" column="APELIDO2"/>
    <property name="dataNacemento" column="DATA_NACEMENTO"/>
```

Actividaes.hbm.xml

```
<class name="Actividades" table="ACTIVIDADES">
    <id name="idActividade" column="ID_ACTIVIDADE" type="int">
        <generator class="identity"/>
    </id>
    <property name="nome" column="NOME"/>
    <property name="numHoras" column="NUM_HORAS"/>
</class>
```

ActividadesAlumnos.hbm.xml

```xml
<class name="ActividadesAlumnos" table="ACTIVIDADES_ALUMNOS">
    <composite-id name="id" class="ActividadesAlumnosId">
        <key-property name="idMatricula" column="ID_MATRICULA"/>
        <key-property name="idActividade" column="ID_ACTIVIDADE"/>
    </composite-id>
    <property name="puntuacion" column="PUNTUACION"/>
</class>
```

ANOTACIÓNS: o equivalente anterior con anotacións sería:

```java
@Entity
@Table(name = "Alumnos", uniqueConstraints = {
    @UniqueConstraint(name = "uk_dni", columnNames = "DNI")
})
public class Alumno {

    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    @Column(name = "DNI", nullable = false)
    private String dni;

    @Column(name = "Nome", nullable = false)
    private String nome;

    @Column(name = "Apelido1", nullable = false)
    private String apelido1;

    @Column(name = "Apelido2")
    private String apelido2;

    @Column(name = "DataNacemento", nullable = false)
    @Temporal(TemporalType.DATE)
    private Date dataNacemento;

    // Getters e setters
}
```

```java
@Entity
@Table(name = "ACTIVIDADES")
public class Actividades implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idActividade")
    private int idActividade;

    @Column(name = "Nome", nullable = false, unique = true)
    private String nome;

    @Column(name = "NumHoras")
    private Integer numHoras;

    // getters e setters
}
```

A clase AlumnoActividade ten unha clave primaria composta polo id do alumno e o id da actividade. Para iso, utilízase a anotación @Embeddable para a clave primaria composta e @EmbeddedId na entidade. Aquí tes un exemplo:

Primeiro, créase unha clase, no noso exemplo AlumnoActividadeId, que representa a clave primaria composta:

```java
@Embeddable
public class ActividadesAlumnosId implements Serializable {
    @Column(name = "idMatricula")
    private String idMatricula;

    @Column(name = "idActividade")
    private int idActividade;

    // getters e setters
}
```

Logo, na clase ActividadesAlumnos, podes usar a anotación @EmbeddedId para indicar que a clave primaria é composta:

```java
@Entity
@Table(name = "ACTIVIDADES_ALUMNOS")
public class ActividadesAlumnos {
    @EmbeddedId
    private ActividadesAlumnosId id;

    @Column(name = "Puntuacion", nullable = false)
    private float puntuacion;

    // getters e setters
}
```

2. Estratexias no mapeo. Entidades e Tipos Valor

Un obxectivo principal de Hibernate é o apoio aos *modelos de obxectos de gran fino*, como un dos requisito máis importante para os mapeos de datos. É unha razón pola cal se traballa con POJOS. O termo de gran fino significa "*máis clases que táboas*". Isto significa que nas filas dunha táboa da base de datos pódese representar máis dun único obxecto e que non todas as clases que teñamos nun proxecto van ter correspondencia unívoca cunha táboa na base de datos.

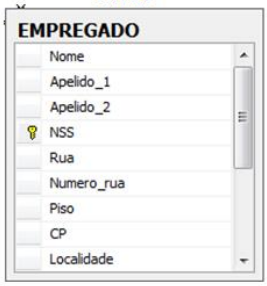
Exemplo: unha clase Cliente con propiedades para o enderezo de facturación (rúa,cidade, código postal, localidade) e para o envío. Pode que estas informacións estean na base de datos nunha única táboa Cliente pero que dende o punto de vista da codificación en Java, pode ser convinte ter unha clase Enderezo con estes datos, ou ter a propiedade dirección de correo electrónico, en vez dunha propiedade de tipo cadea, como unha propiedade dunha clase Email, o que pode engadir un comportamento más sofisticado, ao poder definilo mediante métodos, por exemplo, pode ofrecer un método sendEmail().

No seguinte exemplo, amósanse *dúas clases que se mapean nunha única táboa* da base de datos:

```
public class Empregado  implements java.io.Serializable {
     private String nss;
     private Empregado empregado;
     private String nome;
     private String apelido1;
     private String apelido2;
     private Enderezo enderezo;

 public class Enderezo implements Serializable{
      private String rua;
      private int numeroRua;
      private String piso;
      private String cp;
      private String localidade;
```

Táboa



Este problema de granularidade lévanos a facer unha distinción importante dentro de Hibernate. Si pensamos nun deseño de gran fino ("máis clases que táboas"), unha fila representa a múltiples instancias de diferentes clases. Dado que a identidade dunha táboa da base de datos é implementada mediante a clave primaria, entón algúns obxectos persistentes non teñen a súa propia identidade, é dicir, un dos obxectos representados na fila ten a súa propia identidade (exemplo anterior sería empregado), e outros dependen deste e non teñen un valor identidade (exemplo enderezo). Hibernate fai a distinción esencial seguinte:

- Entidades.
- Tipo Valor (compoñente).

2.1 Entidades

– Un obxecto de tipo de entidade ten unha identidade propia na base de datos (valor de clave principal).

– Unha referencia de obxecto a unha instancia de entidade é persistido como unha referencia na base de datos (un valor de clave externa).

– A entidade ten o seu propio ciclo de vida, e pode existir independentemente de calquera outra entidade.

– O estado persistente dunha entidade consiste en referencias a outras entidades, e a instancias do que en Hibernate se denomina "`value types`" (tipos "valor").

Exemplo: Departamento, Empregado, Proxecto.

2.2 Tipo Valor

– Un obxecto de tipo valor non ten unha identidade de base de datos, senón que pertence a unha instancia dunha entidade e o seu estado persistente incrústase na fila da táboa á que pertence. Obxectos persistentes sen identidade propia (nin táboa propia).

– Os tipos de valor non teñen identificadores ou propiedades identificadoras.

– A vida útil dunha instancia de tipo de valor está limitada pola vida útil da pertenza a instancias de entidade.

– Un tipo de valor non é compatible con referencias compartidas.

Exemplo: Enderezo.

Colección (interface)	Etiqueta de Hibernate	Instancia requirida (clase Java que implementa a interface)	Observacións
java.util.Set	**<set>**	java.util.HashSet.	**Colección non ordenada de valores de obxectos sen repeticións.** A orde dos elementos non é preservada. Non se permiten elementos repetidos.
java.util.SortedSet	**<set>**	java.util.TreeSet	Set non permite ordenar os seus elementos. Para que **os elementos dun conxunto teñan orde**, Java proporciona un tipo SortedSet que herda de Set. A clase que implementar SortedSet chámase TreeSet
java.util.List	**<list>**	java.util.ArrayList	**Colección ordenada de valores de obxectos, admite repetidos**. Respecta a orde dos elementos sempre e cando se especifique unha columna que se utilice como índice.
java.util.Collection	**<bag> ó <idbag>.**	java.util.ArrayList	Java non conta cunha interface ou implementación para bag, aínda que java.util.Collection permite **utilizar a semántica das bags permitindo duplicados** e a orde non se mantén entre os elementos.
java.util.Map	**<map>**	java.util.HashMap	Un mapa de java pode ser utilizado preservando os **pares (clave, valor).**

Os arrays están soportados por Hibernate con `<primitive-array>` (para tipos primitivos Java) e `<array>` (para outros) pero úsanse raramente.

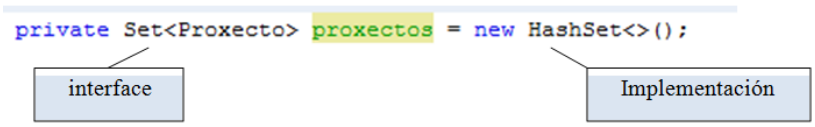
Hibernate recomenda usar interfaces nas declaracións de tipo das coleccións Java para poder realizar o mapeo das coleccións correctamente. O procedemento recomendado é:

- Usar unha interface para declarar o tipo da colección (ex: Set).
- Elixir unha clase que implementar á interface (ex: HashSet) e iniciala na declaración sen esperar a facelo no construtor ou nalgún método setter. Este é o método recomendado.

A sintase para declarar unha propiedade de colección é:

```
privada < Interface > nome_coleccion = new < Implementación> ();
...
/ / Os métodos getter e setter
```

Exemplo:



As coleccións poden conter*: tipos básicos, entidades e compoñentes*. Non se poden crear coleccións de coleccións.

3. Coleccións de compoñentes

Unha colección de compoñentes en Hibernate é unha característica que nos permite manexar un conxunto de obxectos dependentes de unha entidade. Estes obxectos non teñen a súa propia identidade e existen só como parte dun obxecto pai.

Por exemplo, imaxina que tes unha entidade Alumno que ten un conxunto de Telefono como propiedade. Cada Telefono non é unha entidade por si mesma, senón que é parte do Alumno. Neste caso, podes usar unha colección de compoñentes para manexar os teléfonos.

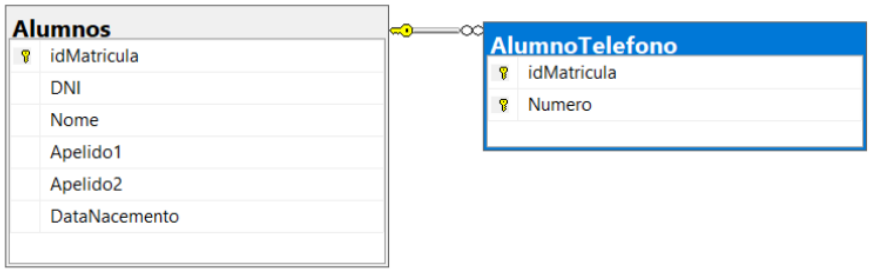
3.1.1 Coleccións set de compoñentes

En Hibernate, unha colección Set é útil por varias razóns:

- Eliminación de duplicados: Unha das principais características dun Set é que non permite elementos duplicados. Isto pode ser moi útil cando queres asegurarte de que cada compoñente na túa colección é único.
- Orde non garantida: Ao contrario dunha lista, un Set non mantén unha orde específica dos seus elementos. Isto pode ser beneficioso en termos de rendemento, xa que non se require manter a orde dos elementos.

Exemplo1 de mapeo:

Na base de datos almacenamos os teléfonos nunha táboa aparte



Con pojo Alumno.java e ficheiro de mapeo Alumno.hbm.xml.

```java
public class Alumnos implements Serializable {
    private String idMatricula;
    private String dni;
    private String nome;
    private String apelido1;
    private String apelido2;
    private Date dataNacemento;
    private Set<String> telefonos;

    // getters e setters
```

```xml
<hibernate-mapping>
    <class name="Alumnos" table="ALUMNOS">
        <id name="idMatricula" column="idMatricula" type="string"/>
        <property name="dni" column="DNI" not-null="true" unique="true"/>
        <property name="nome" column="Nome" not-null="true"/>
        <property name="apelido1" column="Apelido1" not-null="true"/>
        <property name="apelido2" column="Apelido2"/>
        <property name="dataNacemento" column="DataNacemento" not-null="true"/>
        <set name="telefonos" table="AlumnoTelefono">
            <key column="idMatricula"/>
            <element column="Numero" type="string"/>
        </set>
    </class>
</hibernate-mapping>
```

Con pojo Alumno.java e anotacións

```java
@Entity
@Table(name = "ALUMNOS")
public class Alumnos implements Serializable {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    @ElementCollection
    @CollectionTable(name = "AlumnoTelefono", joinColumns = @JoinColumn(name =
"idMatricula"))
    @Column(name = "Numero")
    private Set<String> telefonos;
```

- ✓ Neste código, estamos a definir unha entidade Alumnos que ten unha *colección Set de números de teléfono* (telefonos).
- ✓ Cada *número de teléfono* é *un String e non unha entidade por si mesmo*, polo que utilizamos ***@ElementCollection***.

- ✓ Os números de teléfono almacénanse *nunha táboa separada chamada AlumnoTelefono*, e esta táboa únese á táboa ALUMNOS a través da columna de clave foránea *idMatricula*., polo que utilizamos ***@CollectionTable(name=”AlumnoTelefono”, joinColumns = @JoinColumn(name = "idMatricula"))***
- ✓ Cada *número de teléfono* almacénase na *columna Numero da táboa AlumnoTelefono*, polo que utilizamos ***@Column(name = "Numero")***.

Explicacións das etiquetas para colecciones de elementos tipo valor:

- **@ElementCollection**: Esta anotación utilízase para especificar unha colección de elementos incrustados ou de valores básicos.

  Utilízase cando queremos ter unha colección de elementos que son entidades pero non teñen unha clave primaria.

  A sintaxe xeral é:

```
@ElementCollection
private Collection<TipoElemento> nomeColeccion;
```

  Por exemplo, se temos unha entidade Alumnos e queremos almacenar unha colección de números de teléfono para cada alumno, pero os números de teléfono non son unha entidade por si mesmos.

- **@CollectionTable**: Esta anotación utilízase xunto con @ElementCollection para especificar detalles sobre a táboa onde se almacenarán os elementos da colección. A sintaxe xeral é:

```
@ElementCollection
@CollectionTable(name = "nomeTáboa", joinColumns = @JoinColumn(name =
"ClaveForánea"))
private Collection<TipoElemento> nomeColeccion;
```

  No exemplo, name = “AlumnoTelefono” especifica o nome da táboa onde se almacenarán os números de teléfono.

- **@JoinColumn:** Esta anotación utilízase para especificar a columna que se utilizará para unir a táboa da colección coa táboa da entidade principal (a clave foránea) .

  name = “idMatricula” especifica que a columna idMatricula é a clave foránea.

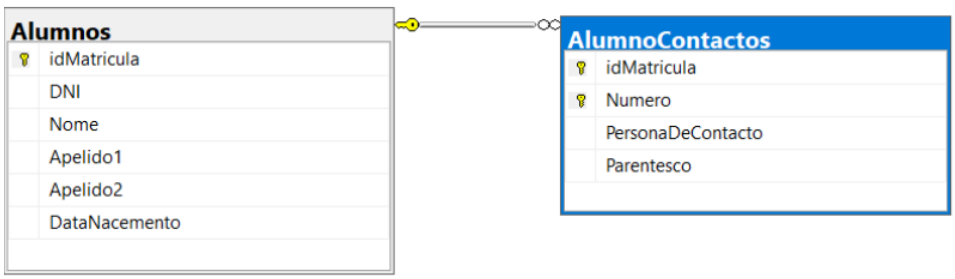
- **@Column:** Esta anotación utilízase para especificar a columna que se utilizará para almacenar os elementos da colección na táboa da colección. A sintaxe xeral é:

```
@Column(name = "nomeColumna")
```

  name = “Numero” especifica que os números de teléfono almacenaranse na columna Numero da táboa AlumnoTelefono.

Exemplo2 de mapeo:

Na base de datos almacenamos os teléfonos dos contactos xunto ao nome e parentesco do contacto nunha táboa aparte:



Con pojo Alumno.java,AlumnoContacto.java e ficheiro de mapeo Alumno.hbm.xml.

AlumnoContactos non é unha entidade, no ten propiedade identificadora e o ciclo de vida depende do ciclo da entidade propietaria Alumnos

```java
public class Alumnos {
    private String idMatricula;
    private String dni;
    private String nome;
    private String apelido1;
    private String apelido2;
    private Date dataNacemento;
    private Set<AlumnoContactos> contactos = new HashSet<>();

    // Getters and setters
```

```java
public class AlumnoContactos {
    private String numero;
    private String personaDeContacto;
    private String parentesco;

    // Getters and setters
    // ...
}
```

```xml
<hibernate-mapping package="POJOS">
    <class name="Alumnos" table="ALUMNOS">
        <id name="idMatricula" type="string" column="IDMATRICULA">
            <generator class="assigned" />
        </id>
        <property name="dni" column="DNI" type="string" />
        <property name="nome" column="NOME" type="string" />
        <property name="apelido1" column="APELIDO1" type="string" />
        <property name="apelido2" column="APELIDO2" type="string" />
        <property name="dataNacemento" column="DATANACEMENTO" type="date" />
        <set name="contactos">
            <key column="IDMATRICULA" not-null="true" />
            <composite-element class="AlumnoContactos">
                <property name="numero" column="NUMERO" type="string" />
                <property name="personaDeContacto" column="PERSONADECONTACTO" type="string"
/>
                <property name="parentesco" column="PARENTESCO" type="string" />
            </composite-element>
        </set>
    </class>
</hibernate-mapping>
```

Equivalente con anotacións:

```java
@Entity
@Table(name = "ALUMNOS")
public class Alumnos {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    @Column(name = "DNI")
    private String dni;

    @Column(name = "Nome")
    private String nome;

    @Column(name = "Apelido1")
    private String apelido1;

    @Column(name = "Apelido2")
    private String apelido2;

    @Column(name = "DataNacemento")
    private Date dataNacemento;

    @ElementCollection
    @CollectionTable(name = "ALUMNOCONTACTOS", joinColumns = @JoinColumn(name =
"idMatricula"))
    private Set<AlumnoContactos> contactos;
}
```

```java
import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class AlumnoContactos {
    @Column(name = "Numero")
    private String numero;

    @Column(name = "PersonaDeContacto")
    private String personaDeContacto;

    @Column(name = "Parentesco")
    private String parentesco;
}
```

- **@ElementCollection** indica que o campo é unha colección de elementos que son instancias dun tipo básico ou un tipo incrustable. @CollectionTable especifica a táboa que se usa para a colección de elementos. **@JoinColumn** especifica a columna de clave foránea.
- **@Embeddable** indica que a clase é un tipo incrustable que se pode incrustar en entidades ou usarse como elementos de colección.

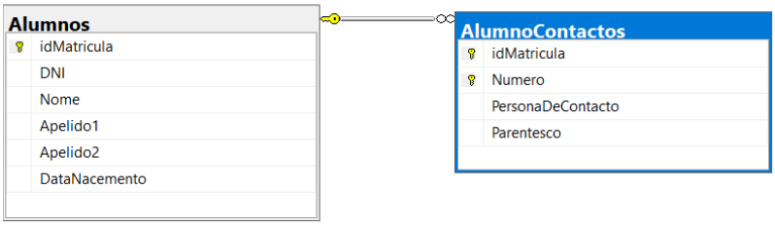
3.1.2 Coleccións map de compoñentes

Unha colección Map en Hibernate pode ser moi útil cando queres asociar un conxunto de elementos cunha entidade, onde cada elemento ten unha clave única. As vantaxes de usar un Map:

- Acceso rápido aos elementos: Grazas ás claves únicas, podes acceder aos elementos dun Map de maneira moi rápida. Isto é especialmente útil cando tes un gran número de elementos na túa colección.
- Orde natural dos elementos: Se usas un TreeMap, os elementos da túa colección estarán ordenados segundo a orde natural das súas claves. Isto pode ser moi útil para mostrar os datos dunha maneira específica.
- Prevención de duplicados: Como as claves dun Map son únicas, non podes ter dous elementos coa mesma clave. Isto pode axudar a previr erros de datos.

Exemplo de mapeo:

Na base de datos almacenamos os teléfonos dos contactos xunto ao nome e parentesco do contacto nunha táboa aparte e imos a mapearlo como unha colección map <clave, valor>



Con pojo Alumno.java,AlumnoContactos.java e ficheiro de mapeo Alumno.hbm.xml.

Agora a clave é o número de telefono e o valor está defindio na clase AlumnoContactos.

```java
public class Alumnos {
    private String idMatricula;
    private String dni;
    private String nome;
    private String apelido1;
    private String apelido2;
    private Date dataNacemento;
    private Map<String, AlumnoContactos> contactos = new HashMap<>();

    // Getters and setters
    // ...
}
```

```java
public class AlumnoContactos {
    private String personaDeContacto;
    private String parentesco;

    // Getters and setters
    // ...
}
```

```xml
<hibernate-mapping>
    <class name="POJOS.Alumnos" table="alumnos">
        <id name="idMatricula" type="string" column="idMatricula">
            <generator class="assigned" />
        </id>
        <property name="dni" column="dni" type="string" />
        <property name="nome" column="nome" type="string" />
        <property name="apelido1" column="apelido1" type="string" />
        <property name="apelido2" column="apelido2" type="string" />
        <property name="dataNacemento" column="dataNacemento" type="date" />
        <map name="contactos">
            <key column="idMatricula" not-null="true" />
            <map-key column="numero" type="string" />
            <composite-element class="POJOS.AlumnoContactos">
                <property name="personaDeContacto" column="personaDeContacto" type="string"
/>

                <property name="parentesco" column="parentesco" type="string" />
            </composite-element>
        </map>
    </class>
</hibernate-mapping>
```

Equivalente con anotacións:

```java
@Entity
@Table(name = "alumnos")
public class Alumnos {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    @Column(name = "dni")
    private String dni;

    @Column(name = "nome")
    private String nome;

    @Column(name = "apelido1")
    private String apelido1;

    @Column(name = "apelido2")
    private String apelido2;

    @Column(name = "dataNacemento")
    private Date dataNacemento;

    @ElementCollection
    @CollectionTable(name = "alumnocontactos", joinColumns = @JoinColumn(name =
"idMatricula"))
    @MapKeyColumn(name = "numero")
    private Map<String, AlumnoContactos> contactos;
}
```

```java
import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class AlumnoContactos {
    @Column(name = "personaDeContacto")
    private String personaDeContacto;

    @Column(name = "parentesco")
    private String parentesco;
}
```

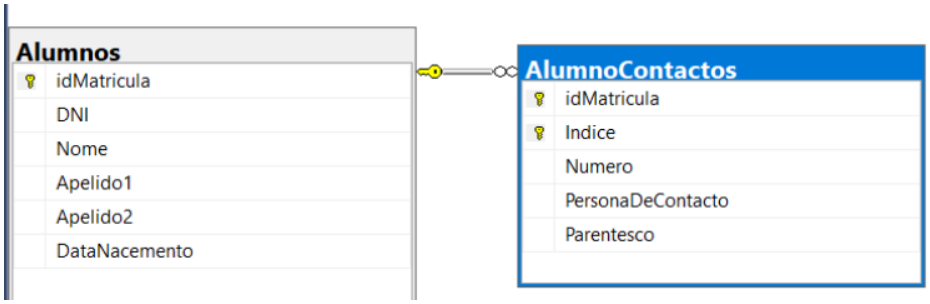
- **@ElementCollection** indica que o campo é unha colección de elementos que son instancias dun tipo básico ou un tipo incrustable.
- **@CollectionTable** especifica a táboa que se usa para a colección de elementos. @JoinColumn especifica a columna que se usa para unirse á táboa de colección.
- **@MapKeyColumn** especifica a columna que se utilizará como clave no Map.
- **@Embeddable** indica que a clase é un tipo incrustable que se pode incrustar en entidades ou usarse como elementos de colección.

3.1.3 Coleccións list de compoñentes

Para mapear unha lista de compoñentes en Hibernate, na táboa da base de datos debe ter un campo de índice (numérico enteiro) adicional para manter a orde dos elementos na lista e este valor o xenará Hibernate. No caso anterior a táboa engadimos o campo índice.

```sql
-- Crea a táboa Alumnos.
CREATE TABLE Alumnos (
    idMatricula CHAR(6) NOT NULL,
    DNI CHAR(9) NOT NULL,
    Nome VARCHAR(100) NOT NULL,
    Apelido1 VARCHAR(100) NOT NULL,
    Apelido2 VARCHAR(100),
    DataNacemento DATE NOT NULL,
    CONSTRAINT pk_alumnos PRIMARY KEY (idMatricula),  -- Define a chave primaria
    CONSTRAINT uk_dni UNIQUE (DNI),  -- Asegura que o DNI é único
    CONSTRAINT ck_dni CHECK (DNI LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][A-Z]')  --
Verifica o formato do DNI
);
```

```sql
-- Crea a táboa AlumnoContactos.
CREATE TABLE AlumnoContactos (
    idMatricula CHAR(6) NOT NULL,  -- Corresponde ao campo idMatricula na clase Alumno
    Indice INT NOT NULL,  -- Corresponde ao índice na lista contactos
    Numero CHAR(9) NOT NULL,
    PersonaDeContacto VARCHAR(50) NOT NULL,
    Parentesco VARCHAR(100) NOT NULL,
    CONSTRAINT PK_ALUMNO_TELEFONO PRIMARY KEY (idMatricula, Indice),  -- Define a chave
primaria
    CONSTRAINT FK_ALUMNO_TELEFONO FOREIGN KEY (idMatricula) REFERENCES ALUMNOS(idMatricula),
-- Define a chave estranxeira
    CONSTRAINT UK_ALUMNO_TELEFONO UNIQUE (idMatricula, Numero)  -- Asegura que o número de
teléfono é único para cada alumno
);
```



Con ficheiro de mapeo Alumno.hbm.xml

```
public class Alumno {

    private String idMatricula;
    private String dni;
    private String nome;
    private String apelido1;
    private String apelido2;
    private Date dataNacemento;
    private List<AlumnoContacto> contactos = new ArrayList<>();

    // getters y setters
}
```

```
public class AlumnoContacto {

    private String numero;
    private String personaDeContacto;
    private String parentesco;

    // getters y setters
}
```

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="POJOS">
    <class name="Alumno" table="dbo.Alumnos">
        <id name="idMatricula" column="idMatricula" type="string"/>
        <property name="dni" column="DNI" type="string"/>
        <property name="nome" column="Nome" type="string"/>
        <property name="apelido1" column="Apelido1" type="string"/>
        <property name="apelido2" column="Apelido2" type="string"/>
        <property name="dataNacemento" column="DataNacemento" type="date"/>
        <list name="contactos">
            <key column="idMatricula"/>
            <list-index column="Indice" base="1"/>
            <composite-element class="AlumnoContacto">
                <property name="numero" column="Numero" type="string"/>
                <property name="personaDeContacto" column="PersonaDeContacto"
type="string"/>
                <property name="parentesco" column="Parentesco" type="string"/>
            </composite-element>
        </list>
    </class>
</hibernate-mapping>
```

Con anotacións:

```
@Entity
@Table(name = "Alumnos")
public class Alumno {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;
    @Column(name = "DNI", unique = true, nullable = false)
    private String dni;
    @Column(name = "Nome", nullable = false)
    private String nome;
    @Column(name = "Apelido1", nullable = false)
    private String apelido1;
    @Column(name = "Apelido2")
    private String apelido2;
    @Column(name = "DataNacemento", nullable = false)
    private Date dataNacemento;
    @ElementCollection
    @CollectionTable(name = "AlumnoContactos", joinColumns = @JoinColumn(name =
"idMatricula"))
    @OrderColumn(name = "Indice")
    @ListIndexBase(1)
    private List<AlumnoContacto> contactos = new ArrayList<>();
    // Getters and setters
}
```

```
@Embeddable
public class AlumnoContacto {
    @Column(name = "Numero", nullable = false)
    private String numero;
    @Column(name = "PersonaDeContacto", nullable = false)
    private String personaDeContacto;
    @Column(name = "Parentesco", nullable = false)
    private String parentesco;
    // Getters and setters
}
```

- **@Embeddable**: Esta anotación indica que a clase AlumnoContacto é un tipo de dato que se pode incrustar noutros obxectos entidade. Neste caso, AlumnoContacto é un tipo de dato que se incrusta na entidade Alumno.
- **@ElementCollection**: Esta anotación indica que a lista contactos é unha colección de elementos que son instancias da clase incrustable AlumnoContacto. Estes elementos non son entidades completas, polo que non teñen un identificador propio.
- **@CollectionTable(name = "AlumnoContactos", joinColumns = @JoinColumn(name = "idMatricula")):** Esta anotación indica que os elementos da colección contactos deben almacenarse nunha táboa separada chamada "AlumnoContactos". A columna "idMatricula" desta táboa está asociada coa entidade Alumno.
- **@OrderColumn(name = "Indice"):** Esta anotación indica que Hibernate debe manter a orde dos elementos na colección contactos nunha columna chamada "Indice". Cando se recupera a entidade Alumno, Hibernate ordenará a colección contactos segundo os valores desta columna.
- **@ListIndexBase(1):** Esta anotación é específica de Hibernate e indica que o índice da lista contactos debe comezar en 1 en lugar de 0. Esta anotación non é parte do estándar JPA.

Hai varias razóns para querer usar unha colección List de compoñentes en Hibernate:

- ✓ Orde conservado: Unha das principais vantaxes de usar unha List é que conserva a orde dos elementos. Isto pode ser útil se a orde dos teus compoñentes é importante para a túa aplicación.
- ✓ Acceso indexado: As listas permiten o acceso indexado aos seus elementos, o que significa que podes recuperar ou modificar un elemento nunha posición específica na lista. Isto pode ser útil se necesitas realizar operacións baseadas no índice.
- ✓ Duplicados permitidos: Ao contrario dos conxuntos (Set), as listas permiten elementos duplicados. Se os teus compoñentes poden ter duplicados e queres conservar todos eles, entón unha lista pode ser a mellor opción.
- ✓ Mapeo máis sinxelo: Hibernate proporciona un mapeo máis sinxelo para as listas a través da anotación @OrderColumn, que permite a Hibernate manter a orde dos elementos na lista.

A elección entre diferentes tipos de coleccións debe basearse nas necesidades específicas da túa aplicación. Se a orde dos elementos non é importante, un conxunto (Set) pode ser máis eficiente. Se necesitas un mapeo chave-valor, entón un mapa (Map) sería a mellor opción.

3.1.4 Coleccións idbag de compoñentes

Pódese usar un IdBag en lugar dun List para almacenar unha colección de compoñentes en Hibernate.

Tanto List como IdBag en Hibernate **usan un índice para cada elemento na colección**. A diferenza principal entre eles é como Hibernate xestiona as operacións de actualización:

- ✓ Un List en Hibernate é un tipo de colección que mantén a orde dos elementos e permite duplicados. Cando usas un List, Hibernate usa a posición do elemento na lista como índice.

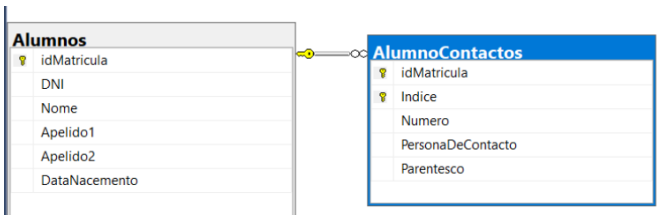
  Este índice é xerado automaticamente por Hibernate e non ten un significado real na túa lóxica de negocio.

  Cando actualizas un elemento nun List, Hibernate necesita eliminar todos os elementos da lista na base de datos e inserilos de novo para manter a orde correcta. Isto pode ser ineficiente se a túa lista é grande.
- ✓ Un IdBag tamén é un tipo de colección que permite duplicados, pero a diferenza dun List, un IdBag usa un identificador único xerado por Hibernate para cada elemento da colección dentro de una entidad. Este identificador é almacenado nunha columna adicional na táboa de relación.

  Cando actualizas un elemento nun IdBag, Hibernate só necesita actualizar o elemento específico na base de datos, o que é máis rápido e eficiente.

Por tanto, se realizas moitas operacións de actualización na túa colección, un IdBag pode ser máis eficiente. Se a orde dos elementos é importante para a túa lóxica de negocio, entón deberías usar un List.

A anterior táboa AlumnoContactos pódese tamén mapear como un idbag de compoñentes:



Con ficheiro de mapeo Alumno.hbm.xml

```java
// Clase Alumno
package POJOS;

import java.util.*;

public class Alumno {
    private String idMatricula;

    // Outros campos aquí...

    private Collection<Contacto> contactos = new ArrayList<>();

    // Getters e setters aquí...
}
```

```java
// Clase Contacto
package POJOS;

public class Contacto {
    private String numero;
    private String personaDeContacto;
    private String parentesco;

    // Getters e setters aquí...
}
```

```xml
<hibernate-mapping package="POJOS">
    <class name="Alumno" table="dbo.Alumnos">
        <id name="idMatricula">
            <generator class="assigned"/>
        </id>
        <!-- Mapeo de outros campos aquí -->
        <idbag name="contactos" table="dbo.AlumnoContactos" lazy="true">
            <collection-id column="Indice" type="long" generator="sequence"/>
            <key column="idMatricula"/>
            <composite-element class="Contacto">
                <property name="numero" column="Numero"/>
                <property name="personaDeContacto" column="PersonaDeContacto"/>
                <property name="parentesco" column="Parentesco"/>
            </composite-element>
        </idbag>
    </class>
</hibernate-mapping>
```

Con anotacións:

```java
@Entity
@Table(name = "dbo.Alumnos")
public class Alumno {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    // Outros campos aquí...

    @ElementCollection
    @CollectionTable(name = "dbo.AlumnoContactos", joinColumns = @JoinColumn(name =
"idMatricula"))
    @OrderColumn(name="Indice")
    @GeneratedValue(generator="sequence-gen")
    @GenericGenerator(name="sequence-gen", strategy="sequence")
    private Collection<Contacto> contactos = new ArrayList<>();

    // Getters e setters aquí...
}
```

```java
@Embeddable
public class Contacto {
    @Column(name = "Numero")
    private String numero;

    @Column(name = "PersonaDeContacto")
    private String personaDeContacto;

    @Column(name = "Parentesco")
    private String parentesco;

    // Getters e setters aquí...
}
```

- **@ElementCollection:** Esta anotación é usada para especificar unha colección de instancias dun tipo básico ou embeddable. Neste caso, estás a usar @ElementCollection para indicar que a colección contactos na clase Alumno é unha colección de instancias da clase embeddable Contacto.
- **@CollectionTable:** Esta anotación é usada xunto con @ElementCollection para definir a táboa que será usada para almacenar a colección de elementos. Neste caso, @CollectionTable(name = "dbo.AlumnoContactos", joinColumns = @JoinColumn(name = "idMatricula")) indica que a colección contactos será almacenada na táboa dbo.AlumnoContactos e que a columna idMatricula será usada como chave foránea.
- **@OrderColumn**: Esta anotación é usada para especificar a columna que se usa para manter a orde dos elementos nunha lista. Neste caso, @OrderColumn(name="Indice") indica que a columna Indice será usada para manter a orde dos contactos na lista.
- **@GeneratedValue e @GenericGenerator:** Estas anotacións son usadas para definir a estratexia de xeración de valores para a columna do índice. @GenericGenerator(name="sequence-gen", strategy="sequence") define un xerador de valores co nome "sequence-gen" que usa a estratexia "sequence". Logo, @GeneratedValue(generator="sequence-gen") indica que se debe usar este xerador de valores para xerar os valores para a columna do índice.

3.1.5 Coleccións bag de compoñentes

En Hibernate, un Bag é unha colección non ordenada de elementos que permite duplicados, mentres que un Set é unha colección que non permite duplicados.

A razón pola que se pode preferir usar un Bag en lugar dun Set é que os Bags son máis eficientes para certas operacións. Por exemplo, cando se engade un elemento a un Bag, Hibernate non necesita verificar se o elemento xa existe na colección, o que fai que a operación de inserción sexa máis rápida. Ademais, os Bags son máis flexibles porque permiten duplicados.

Por outra banda, os Sets son máis eficientes para operacións de busca, xa que non permiten duplicados e, polo tanto, as operacións de busca son máis rápidas. Non obstante, as operacións de inserción e eliminación son máis lentas porque Hibernate ten que verificar a existencia de cada elemento.

En resumo, a elección entre un Bag e un Set depende das necesidades específicas da aplicación. Si se necesita unha colección que permita duplicados e que sexa eficiente para insercións, entón un Bag pode ser a mellor opción. Se necesitas unha colección que non permita duplicados e que sexa eficiente para buscas, entón un Set pode ser a mellor opción

Exemplo: gardar as aflicións dos alumnos e o seu nivel de interese

```
CREATE TABLE AlumnoAficiones (
    idMatricula CHAR(6)  NOT NULL,
    nomeAficion VARCHAR(40),
    nivelInterese tinyint,
    PRIMARY KEY (idMatricula, nomeAficion),
    FOREIGN KEY (idMatricula) REFERENCES Alumnos(idMatricula)
);
```

Con ficheiro de mapeo Alumno.hbm.xml

```
// Clase Alumno
public class Alumno {
    private String idMatricula;  // Corresponde á columna idMatricula na táboa Alumnos
    private Collection<Aficion> aficiones = new ArrayList<>();  // Unha colección Bag de
Aficion

    // Outros campos e os seus getters e setters aquí
}
```

```java
// Clase Aficion
public class Aficion {
    private String nomeAficion;  // Corresponde á columna nomeAficion na táboa AlumnoAficiones
    private String nivelInterese;  // Corresponde á columna nivelInterese na táboa AlumnoAficiones

    // Getters e setters
```

```xml
<hibernate-mapping>
    <class name="POJOS.Alumno" table="dbo.Alumnos">
        <id name="idMatricula">
            <generator class="assigned"/>
        </id>
        <!-- Mapeo de outros campos aquí -->
        <bag name="aficiones" table="dbo.AlumnoAficiones">
            <key column="idMatricula"/>  <!-- Define a chave estranxeira -->
            <composite-element class="POJOS.Aficion">  <!-- Define a clase Aficion que se usa para os elementos da colección -->
                <property name="nomeAficion" column="nomeAficion"/>  <!-- Define o campo nomeAficion na clase Aficion -->
                <property name="nivelInterese" column="nivelInterese"/>  <!-- Define o campo nivelInterese na clase Aficion -->
            </composite-element>
        </bag>
    </class>
</hibernate-mapping>
```

Con anotacións:

```java
public class Alumno {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    @ElementCollection
    @CollectionTable(name = "AlumnoAficiones", joinColumns = @JoinColumn(name = "idMatricula"))
    private Collection<Aficion> aficiones = new ArrayList<>();

    // Outros campos e os seus getters e setters aquí
    // ...
}
```

```java
// Clase Aficion
@Embeddable
public class Aficion {
    @Column(name = "nomeAficion")
    private String nomeAficion;

    @Column(name = "nivelInterese")
    private String nivelInterese;

    // Getters e setters
    // ...
}
```

- **@Embeddable**: Esta anotación indica que a clase na que se usa pode ser incluída como colección de compoñentes dunha entidade. Neste caso, a clase Aficion está marcada como @Embeddable.
- **@ElementCollection**: Esta anotación é usada para especificar unha colección de instancias dun tipo básico ou embeddable. Neste caso, estás a usar @ElementCollection para indicar que a colección aficiones na clase Alumno é unha colección de instancias da clase embeddable Aficion.
- **@CollectionTable**: Esta anotación é usada xunto con @ElementCollection para definir a táboa que será usada para almacenar a colección de elementos. Neste caso, @CollectionTable(name = "AlumnoAficiones", joinColumns = @JoinColumn(name = "idMatricula")) indica que a colección aficiones será almacenada na táboa AlumnoAficiones e que a columna idMatricula será usada como clave foránea.

3.1.6 Coleccións sortedSet y sortedMap de compoñentes

En Hibernate, podes usar varias coleccións ordenadas para o mapeo e son as seguintes:

- **java.util.SortedSet**: Un SortedSet é un conxunto que proporciona unha orde total dos seus elementos. Cando iteras a través do conxunto, os elementos son devoltos en orde ascendente.
- **java.util.SortedMap**: Un SortedMap é un mapa que mantén unha orde total das súas claves. Cando iteras a través do mapa, as entradas son devoltas en orde ascendente das claves.

Exemplo: gardar as aflicións dos alumnos e o seu nivel de interese.

```sql
CREATE TABLE AlumnoAficiones (
    idMatricula CHAR(6)  NOT NULL,
    nomeAficion VARCHAR(40),
    nivelInterese tinyint,
    PRIMARY KEY (idMatricula, nomeAficion),
    FOREIGN KEY (idMatricula) REFERENCES Alumnos(idMatricula)
);
```

Con ficheiro de mapeo Alumno.hbm.xml

Impleméntase como unha colección set e cando se recupera a colección ven ordenada polo nome da afición.

```java
public class Alumnos {
    private String idMatricula;
    private String DNI;
    private String Nome;
    private String Apelido1;
    private String Apelido2;
    private Date DataNacemento;
    private SortedSet<AlumnoAficiones> aficiones = new TreeSet<>();

    // Getters e setters
}
```

```java
public class AlumnoAficiones implements Comparable<AlumnoAficiones> {
    private String nomeAficion;
    private int nivelInterese;

    // Implementación de compareTo para ordenar por nomeAficion
    public int compareTo(AlumnoAficiones o) {
        return this.nomeAficion.compareTo(o.getNomeAficion());
    }

    // Getters e setters
}
```

Neste exemplo, o método compareTo compara o nomeAficion do obxecto actual co nomeAficion do obxecto pasado como argumento. Se o nomeAficion do obxecto actual é lexicograficamente menor, o método devolve un número negativo. Se é igual, devolve cero. E se é maior, devolve un número positivo. Isto resulta nunha orde alfabética dos elementos na colección aficions.

```xml
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="POJOS">
    <class name="Alumnos" table="dbo.Alumnos" catalog="Colegio">
        <id name="idMatricula" type="string">
            <column name="idMatricula" />
            <generator class="assigned" />
        </id>
        <!-- Outros campos -->
        <set name="aficiones" table="dbo.AlumnoAficiones" sort="natural">
            <key>
                <column name="idMatricula" />
            </key>
            <composite-element class="AlumnoAficiones">
                <property name="nomeAficion" type="string">
                    <column name="nomeAficion" />
                </property>
                <property name="nivelInterese" type="integer">
                    <column name="nivelInterese" />
                </property>
            </composite-element>
        </set>
    </class>
```

Neste ficheiro de mapeo, AlumnoAficiones está mapeado como un SortedSet de compoñentes na clase Alumnos. O atributo sort="natural" indica que a colección aficiones debe ser ordenada naturalmente, o que significa que se usará a implementación compareTo da clase AlumnoAficiones.

A clase AlumnoAficiones implementa a interface Comparable, que require que se defina un método compareTo. Este método determina a orde dos elementos na colección. Neste caso, a colección aficiones se ordene polo nome da afición en orde alfabética.

Con Anotacións:

```java
import javax.persistence.*;
import org.hibernate.annotations.SortNatural;
import java.util.SortedSet;
import java.util.TreeSet;

@Entity
@Table(name = "Alumnos", schema = "dbo", catalog = "Colegio")
public class Alumnos {
    @Id
    @Column(name = "idMatricula")
    private String idMatricula;

    // Outros campos

    @ElementCollection
    @CollectionTable(name = "AlumnoAficiones", joinColumns = @JoinColumn(name =
"idMatricula"))
    @SortNatural
    private SortedSet<AlumnoAficiones> aficiones = new TreeSet<>();
}
```

```java
@Embeddable
public class AlumnoAficiones implements Comparable<AlumnoAficiones> {
    @Column(name = "nomeAficion")
    private String nomeAficion;

    @Column(name = "nivelInterese")
    private int nivelInterese;

    // Implementación de compareTo para ordenar por nomeAficion
    public int compareTo(AlumnoAficiones o) {
        return this.nomeAficion.compareTo(o.getNomeAficion());
    }

    // Getters e setters
}
```

- **@ElementCollection:** Esta anotación é usada para especificar unha colección de elementos incrustados. Un elemento incrustado é un obxecto que non ten a súa propia identidade. Neste caso, AlumnoAficiones é un elemento incrustado que forma parte da entidade Alumnos.
- **@CollectionTable**: Esta anotación é usada para especificar a táboa asociada cunha colección de elementos incrustados. Aquí, name = "AlumnoAficiones" especifica o nome da táboa que contén os elementos da colección aficiones. joinColumns = @JoinColumn(name = "idMatricula") especifica a columna de unión entre a táboa Alumnos e a táboa AlumnoAficiones. Neste caso, a columna de unión é idMatricula.
- **@SortNatural**: Esta anotación é usada para indicar que a colección debe ser ordenada de forma natural. Isto significa que se usará a implementación do método compareTo da interface Comparable na clase AlumnoAficiones para ordenar os elementos da colección aficiones.

Imos facer o mesmo como unha colección map e cando se recupera a colección ven ordenada polo nome da afición.

```java
import java.util.Date;
import java.util.SortedMap;
import java.util.TreeMap;

public class Alumnos {
    private String idMatricula;
    private String DNI;
    private String Nome;
    private String Apelido1;
    private String Apelido2;
    private Date DataNacemento;
    private SortedMap<String, Integer> aficiones = new TreeMap<>();

    // Getters e setters
}
```

Nesta versión da clase Alumnos, a variable aficiones é un SortedMap onde a clave é un String que representa o nomeAficion e o valor é un Integer que representa o nivelInterese. Un SortedMap garante que as entradas están ordenadas polas claves, neste caso, por nomeAficion.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="POJOS">
    <class name="Alumnos" table="dbo.Alumnos" catalog="Colegio">
        <id name="idMatricula" type="string">
            <column name="idMatricula" />
            <generator class="assigned" />
        </id>
        <!-- Outros campos -->
        <map name="aficiones" table="dbo.AlumnoAficiones" sort="natural">
            <key>
                <column name="idMatricula" />
            </key>
            <map-key type="string" column="nomeAficion" />
            <element type="integer" column="nivelInterese" />
        </map>
    </class>
</hibernate-mapping>
```

Neste ficheiro de mapeo, aficiones está mapeado como un SortedMap onde a clave é nomeAficion e o valor é nivelInterese. O atributo sort="natural" indica que o mapa debe ser ordenado naturalmente, o que significa que as claves serán ordenadas en orde alfabética.