

Mapeo de clases

Repositorios de Información



POJO (Plain Old Java Object)

- Las clases que necesitan ser persistentes son clases java planas (java beans)
- Tienen que respetar un mínimo convenio de nombrado
 - Setters/getters, constructor sin parámetros, etc.
- La información necesaria para persistencia se añade en forma de metadatos
 - Annotations @
 - xml

POJO Ejemplo (entidad)

```
@Entity
public class Customer implements Serializable {
    private Long id;
    private String name;
    private Address address;
    private Collection<Order> orders = new HashSet();
    private Set<PhoneNumber> phones = new HashSet();

    public Customer() {}    // No-arg constructor

    @Id @GeneratedValue // property access is used
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
}
```

POJO Ejemplo (entidad)

```
@OneToMany
public Collection<Order> getOrders() {
    return orders;
}
public void setOrders(Collection<Order> orders) {
    this.orders = orders;
}
@ManyToMany
public Set<PhoneNumber> getPhones() {
    return phones;
}
public void setPhones(Set<PhoneNumber> phones) {
    this.phones = phones;
}

// Business method to add a phone number to the customer
public void addPhone(PhoneNumber phone) {
    this.getPhones().add(phone);
    // Update the phone entity instance to refer to this customer
    phone.addCustomer(this);
}
```

POJO Ejemplo (Value Object)

```
@Embeddable
public class EmploymentPeriod implements Serializable {
    private Date start;
    private Date end;

    public EmploymentPeriod() {}

    public EmploymentPeriod(Date start, Date end){
        this.start = (Date)start.clone();
        this.end = (Date)end.clone();
    }

    @Column(nullable=false)
    public Date getStartDate() { return (Date)start.clone(); }
    protected void setStartDate(Date start) {
        this.start = start;
    }

    public Date getEndDate() { return (Date)end.clone(); }
    protected void setEndDate(Date end) {
        this.end = end;
    }
}
```

No lleva @Id

Tipo de acceso (field, property) igual al de la clase que lo incluye

Posición de @Id

Acceso por getters/setters

Acceso por atributos

```
@Entity
public class Averia {

    @Id private Long id;
    ...
}
```

```
@Entity
public class Averia {

    private Long id;
    ...

    @Id public Long getId() {
        return id;
    }
    ...
}
```

Código ejecutado por el mapeador para cargar la clase en memoria

```
Averia a = new Averia();
a.id = rs.getLong("id");
...
```

```
Averia a = new Averia();
a.setId( rs.getLong("id") );
...
```

POJOs en JPA

- Constructor sin parámetros obligatorio
- Identificador
 - Preferiblemente no tipos básicos (`int`, `long`, etc.), mejor tipos nullables (`Integer`, `Long`, etc.)
 - Mejor no claves compuestas
 - Se corresponderán con la clave primaria de la tabla
- Getters y Setters (`get/set/is`) para cada atributo
 - si acceso por getters/setters
 - pueden ser privados
 - JPA puede usar los setters al cargar un objeto para ajustar sus atributos
- Colecciones para asociaciones `many`
 - Puede ser `Set<T>`, `List<T>`, `Map<T>` o `Collection<T>`
 - Setters y getters pueden ser privados

Persistencia de campos en JPA

- **Tipos JDK:** Mapeo por defecto, el mapeador ya sabe cómo hacerlo
- **Campos de otro tipo:**
 - **Referencia a ValueType:**
 - todos los campos a la misma tabla
 - Clases `@Embeddable`, o atributos `@Embedded`
 - **Referencia a Entidad:** son relaciones, no campos. FK a la tabla de `@Entity`
 - **Resto de casos, serialización**
 - Debe implementar `Serializable`



Metadatos en annotations

- Añadidas al lenguaje Java desde la versión 5
- `@Entity`, `@Embeddable`, `@Id`, etc.
- Añadidas sobre la clase a mapear
- Cómodas para el programador
 - Se compilan, detección temprana de errores

Metadatos en XML



- En fichero `orm.xml`
- En `persistence.xml`
 - Fichero referenciados desde `persistence.xml`
- XML revoca las indicaciones de Annotations
 - En despliegue pueden se pueden ajustar rendimientos sin tocar código fuente

Metadatos xml, ejemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
  version="2.1">

  <entity name="MedioPago" class="uo.ri.domain.entities.MedioPago">
    <table name="TMediosPago" catalog="" schema="" />
    <inheritance strategy="SINGLE_TABLE" />
    <attributes>
      <id name="id">
        <generated-value strategy="IDENTITY" generator="" />
      </id>
      <many-to-one name="cliente" target-entity="uo.ri.domain.entities.Cliente"
        fetch="EAGER" optional="true" />
      <one-to-many name="cargos" target-entity="uo.ri.domain.entities.Cargo"
        fetch="LAZY" mapped-by="medioPago" orphan-removal="false" />
    </attributes>
  </entity>
```



Categorías de anotaciones

- Entity
- Database Schema
- Identity
- Direct Mappings
- Relationship mappins
- Composition
- Inheritance
- Locking
- Lifecycle
- Entity Manager
- Queries

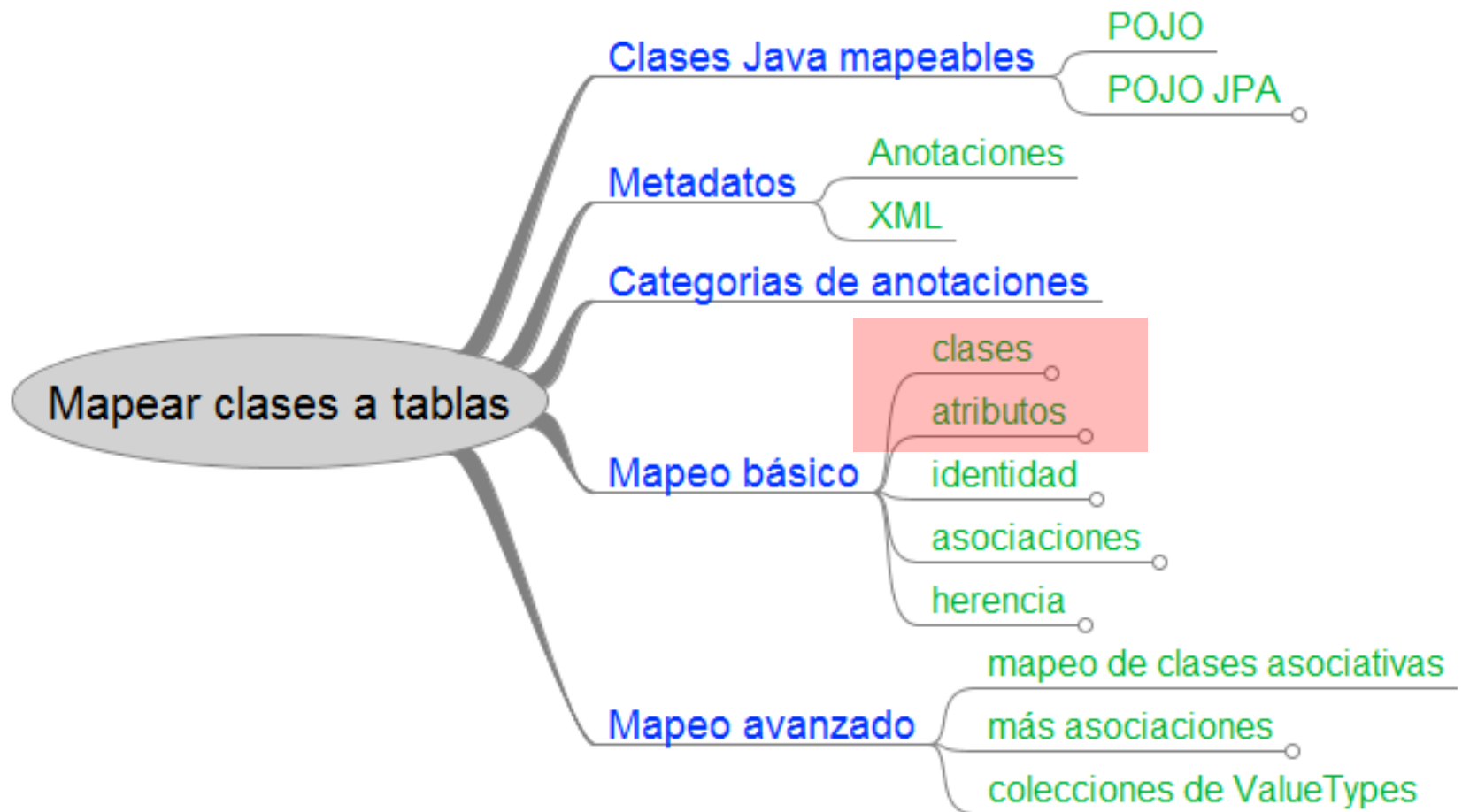
Anotaciones por categoría

Category	Annotations
Entity	<u>@Entity</u> @AccessType
Database Schema Attributes	<u>@Table</u> <u>@SecondaryTable</u> <u>@SecondaryTables</u> <u>@Column</u> <u>@JoinColumn</u> <u>@JoinColumns</u> <u>@PrimaryKeyJoinColumn</u> <u>@PrimaryKeyJoinColumns</u> <u>@JoinTable</u> <u>@UniqueConstraint</u>
Identity	<u>@Id</u> <u>@IdClass</u> <u>@EmbeddedId</u> <u>@GeneratedValue</u> <u>@SequenceGenerator</u> <u>@TableGenerator</u>

Direct Mappings	<u>@Basic</u> <u>@Enumerated</u> <u>@Temporal</u> <u>@Lob</u> <u>@Transient</u>
Relationship Mappings	<u>@OneToOne</u> <u>@ManyToOne</u> <u>@OneToMany</u> <u>@ManyToMany</u> <u>@MapKey</u> <u>@OrderBy</u> @OrderColumn

Anotaciones por categoría

Category	Annotations	Lifecycle Callback Events		
Composition	<u>@Embeddable</u> <u>@Embedded</u> <u>@ElementCollection</u> <u>@AttributeOverride</u> <u>@AttributeOverrides</u> <u>@AssociationOverride</u> <u>@AssociationOverrides</u>	<u>@PrePersist</u> <u>@PostPersist</u> <u>@PreRemove</u> <u>@PostRemove</u> <u>@PreUpdate</u> <u>@PostUpdate</u> <u>@PostLoad</u> <u>@EntityListeners</u> <u>@ExcludeDefaultListeners</u> <u>@ExcludeSuperclassListeners</u>		
Inheritance	<u>@Inheritance</u> <u>@DiscriminatorColumn</u> <u>@DiscriminatorValue</u> <u>@MappedSuperclass</u> <u>@AssociationOverride</u> <u>@AssociationOverrides</u> <u>@AttributeOverride</u> <u>@AttributeOverrides</u>	Entity Manager	<u>@PersistenceUnit</u> <u>@PersistenceUnits</u> <u>@PersistenceContext</u>	Annotations <u>@PersistenceContexts</u> <u>@PersistenceProperty</u>
Locking	<u>@Version</u>		Queries	<u>@NamedQuery</u> <u>@NamedQueries</u> <u>@NamedNativeQuery</u> <u>@NamedNativeQueries</u> <u>@QueryHint</u> <u>@ColumnResult</u> <u>@EntityResult</u> <u>@FieldResult</u> <u>@SqlResultSetMapping</u> <u>@SqlResultSetMappings</u>



Entidades

- Una entidad representa un concepto del dominio
- Puede estar asociada con otras entidades
- Su ciclo de vida es independiente
- Debe tener una clave primaria

```
@Entity
@Table(name="EMP")
public class Employee implements Serializable {
    ...
}
```

Entidades

- @Entity
 - Marca una clase como entidad
- @Table (opcional)
 - Indica la tabla en BBDD

Attribute	Required	Description
name		String
catalog		String
schema		String
uniqueConstraints		@UniqueConstraint .

@Column

```
@Entity
@Table(name = "MESSAGES")
public class Message {
    @Id @GeneratedValue
    @Column(name = "MESSAGE_ID")
    private Long id;
```

- Condiciona la generación de DDL
- Por defecto (sin @Column) cada atributo es un campo en tabla con mismo nombre

```
@Entity
@SecondaryTable(name="EMP_SAL")
public class Employee implements Serializable {
    ...
    @Column(name="SAL", table="EMP_SAL")
    private Long salary;
    ...
}
```

@Column, atributos

Attribute	Required	Description
<code>name</code>		De la comuna en la tabla
<code>unique</code>		Default: <code>false</code> . Estable un índice único en la columna
<code>nullable</code>		Default: <code>true</code> . ¿El campo admite nulos?
<code>insertable</code>		Default: <code>true</code> . Estable si la columna aparecerá en sentencias INSERT generadas
<code>updatable</code>		Default: <code>true</code> . ¿Incluido en SQL UPDATE?
<code>columnDefinition</code>		Default: <code>empty String</code> . Fragmento SQL que se empleará en el DDL para definir esta columna.
<code>table</code>		Default: Todos los campos se almacenan en una única table (see @Table). Si la columna se asocia con otra tabla (see @SecondaryTable), nombre de la otra table especificado en <code>@SecondaryTable</code>
<code>length</code>		Default: 255 para String. Longitud de los campos string.
<code>precision</code>		Default: 0 (sin decimales). Cantidad de decimales.
<code>scale</code>		Default: 0.

@Basic

<u>FetchType</u>	<u>fetch</u> (Opcional) LAZY EAGER Default EAGER.
boolean	<u>optional</u> (Optional) Define si el campo puede ser null.

- Define cómo el mapeador se comportará con respecto al campo
- Aplicable a:

Tipos primitivos, wrappers de primitivos, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], Character[], enums, y Serializable

@Enumerated

- Cómo se salvan los valores enumerados
 - EnumType.ORDINAL
 - EnumType.STRING

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}  
public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}
```

```
@Entity  
public class Employee {  
    ...  
    public EmployeeStatus getStatus() {  
        ...  
    }  
  
    @Enumerated(STRING)  
    public SalaryRate getPayScale() {  
        ...  
    }  
    ...  
}
```

En BDD se creará un campo tipo INTEGER o VARCHAR

@Temporal

- Matiza el formato final de los campos `java.util.Date` y `java.util.Calendar`
 - Opciones: `DATE`, `TIME`, `TIMESTAMP`

```
@Entity
public class Employee {
    ...
    @Temporal (DATE)
    protected java.util.Date startDate;
    ...
}
```

Value Types

- Representan información adicional, no conceptos principales de dominio
- Se suelen presentar como atributos de una entidad o como composiciones (UML)
- Su ciclo de vida depende enteramente de la entidad que las posee
- No pueden tener referencias entrantes

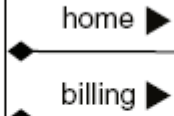
@Embeddable

```
@Embeddable
public class EmploymentPeriod {
    java.sql.Date startDate;
    java.sql.Date endDate;
    ...
}
```

- Marca una clase como ValueType
- Se pueden configurar las propiedades (o atributos) con etiquetas:
 - @Basic, @Column, @Lob, @Temporal, @Enumerated

Ejemplo

User
firstname : String
lastname : String
username : String
password : String
email : String
ranking : int
admin : boolean



Address
street : String
zipcode : String
city : String

```
public class Address {

    private String street;
    private String zipcode;
    private String city;

    public Address() {}

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    public String getZipcode() { return zipcode; }
    public void setZipcode(String zipcode) {
        this.zipcode = zipcode; }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
```

<< Table >> USERS
FIRSTNAME
LASTNAME
USERNAME
PASSWORD
EMAIL
...
HOME_STREET
HOME_ZIPCODE
HOME_CITY
BILLING_STREET
BILLING_ZIPCODE
BILLING_CITY

Component
Columns

Component
Columns

Caso particular

*Si hay más de un atributo
de la misma clase
ValueType ...*

```
@Entity
@Table(name = "USERS")
public class User {
    ...
    @Embedded
    private Address homeAddress;

    @Embedded
    private Address billingAddress;
    ...
}
```

*Si hay más de un VT del mismo tipo en
una entidad hay que forzar los nombres
de las columnas ya que si no se repiten
en el DDL*

```
@Entity
@Table(name = "USERS")
public class User {
    ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "street", column = @Column(name="HOME_STREET") ),
        @AttributeOverride(name = "zipcode", column = @Column(name="HOME_ZIPCODE") ),
        @AttributeOverride(name = "city", column = @Column(name="HOME_CITY") )
    })
    private Address homeAddress;
    ...
}
```

@Lob, @Transient

■ @Lob

```
@Entity
public class Employee implements Serializable {
    ...
    @Lob
    @Basic(fetch=LAZY)
    @Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
    protected byte[] pic;
    ...
}
```

■ @Transient

```
@Entity
public class Employee {
    @Id int id;
    @Transient Session currentSession;
    ...}
```



Identity vs equality

- Java identity
- Object equality
- Database identity
 - `a.getId().equals(b.getId())`
 - clave primaria de la tabla
 - Se mapean con la etiqueta `@Id`
 - Por ello todas las clases **Entidad** deben tener `@Id`entificador
- No siempre serán iguales las tres identidades
 - El periodo de tiempo que sí lo son se le denomina "Ámbito de identidad garantizada", o "Ámbito de persistencia"

Tipos de claves (en BDD)

- Claves candidatas
- Claves naturales
- Claves artificiales (subrogadas)

¿Cuál es mejor para formar la clave primaria en la tabla?

Clave candidata

- Campos (o combinaciones) que permiten determinar de forma única una fila
- Condiciones de una clave
 - Nunca puede ser NULL
 - Cada fila es una combinación única
 - Nunca puede cambiar
- Si hay varias se escogería solo una, las otras son UNIQUE
- Se forman con una sola o combinaciones de propiedades
- Si no hay ninguna está mal el diseño

Claves naturales

- Tienen significado en el contexto de uso (para el usuario: las entiende y las maneja)
 - DNI
 - N° de la SS
- La experiencia demuestra que causan problemas a largo plazo si se usan como claves primarias en tablas
 - ¿Siempre son NOT-NULL?
 - ¿Nunca van a cambiar?
 - ¿Nunca se van a repetir?

¿Y si nos equivocamos al dar el alta?, luego no se puede cambiar ...

Claves artificiales (surrogate keys)

- Sin significado en el contexto
- Siempre generadas por el sistema
- Varias estrategias de generación
 - AUTO → según BBDD
 - IDENTITY → tipo especial en algunas BBDD
 - SEQUENCE
 - TABLE

generan `int`, `long` o `short`

Ver documentación de referencia

Estrategia recomendable

- Usar **siempre** claves artificiales como claves primarias
 - Excepto en el caso de BBDD legacy
- Tipo **Long** suele ser suficiente e indexa de forma eficiente
- La clave natural se hace UNIQUE
 - La que en el modelo de dominio define la identidad
 - Sobre identidad se define **equals()** y **hashCode()**

@Id

```
@Entity
public class Employee implements Serializable {
    @Id @GeneratedValue
    public Long getId() { return id;}
    ...
}
```

- Señala el atributo que forma clave en la tabla
- Clave sencilla: Una @Id
- Clave compuesta:
 - múltiples @Id y una @IdClass, o
 - una @EmbeddedId

```
@IdClass(EmployeePK.class)
@Entity
public class Employee {
    @Id String empName;
    @Id Date birthDay;
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    EmployeePK primaryKey;

    public Employee() { }

    @EmbeddedId
    public EmployeePK getPrimaryKey() {
        return primaryKey;
    }

    public void setPrimaryKey(EmployeePK pk) {
        primaryKey = pk;
    }
}
```

```

@Entity
public class Employee implements Serializable {
    @Id @GeneratedValue
    public Long getId() { return id;}
    ...
}

```

```

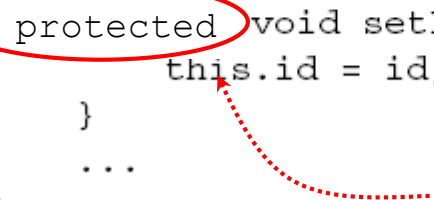
public class Category {
    private Long id;
    ...
    public Long getId() {
        return this.id;
    }
}

```

```

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

```



Si se pone @Id en getters:

La clave debe ser inmutable, una vez asignada no se puede cambiar.

JPA usa el setter cuando se carga en memoria.

*No debe ser público y no puede ser privado → **protected***

@GeneratedValue

- Indica que la clave no es asignada por el programa sino generada por el sistema. Varias estrategias posibles

Attribute	Required	Description
strategy		<p>Default: <code>GenerationType.AUTO</code>.</p> <ul style="list-style-type: none">• <code>IDENTITY</code> – Usa database identity column• <code>AUTO</code> – Usa estategia por defecto de la BDD• <code>SEQUENCE</code> (see @SequenceGenerator)• <code>TABLE</code> – Emplea una table como fuente de claves (see @TableGenerator)
generator		<p>String, el nombre relaciona el generador caracterizado con @SequenceGenerator o @TableGenerator</p>



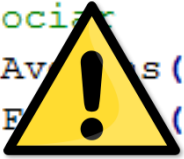
Asociaciones UML implementadas en Java

*Fundamental mantener las **referencias cruzadas***

```
Factura f = ...
Averia a = ...

// asociar
f.getAverias().add( a );
a.setFactura( f );


// desasociar
f.getAverias().remove( a );
a.setFactura( null );
```



```
Factura f = ...
Averia a = ...

// asociar
Association.Facturar.link(a, f);

// desasociar
Association.Facturar.unlink(a, f);
```

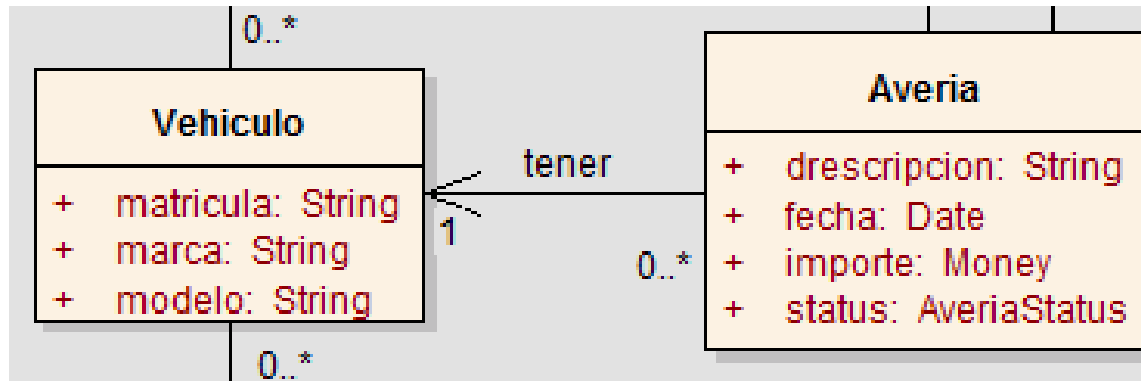


Práctica recomendada: Usar una clase específica

Multiplicidad en JPA

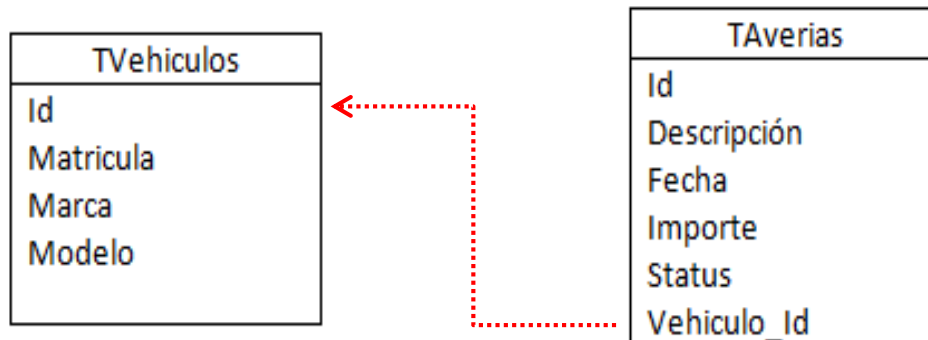
- one-to-one
- many-to-many
- one-to-many
- many-to-one
 - son direccionales, ésta es la inversa de una one-to-many

Unidireccional muchos a uno

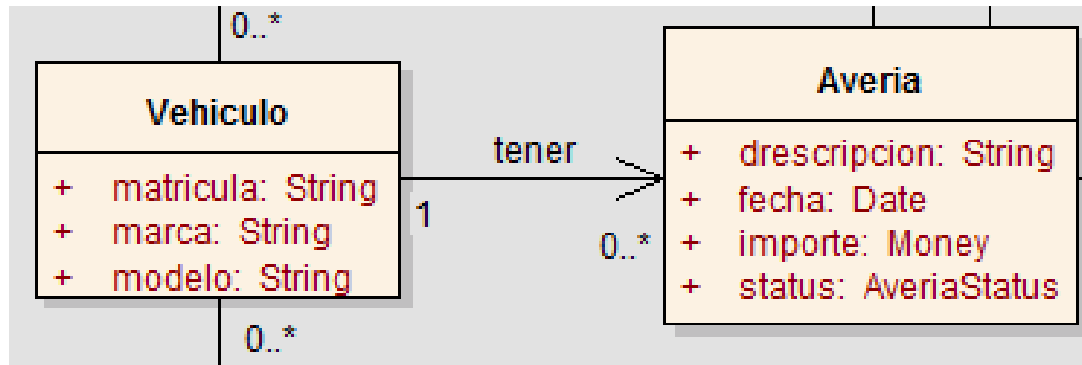


```
public class Vehiculo {  
    ...  
}
```

```
public class Averia {  
    ...  
    @ManyToOne  
    private Vehiculo vehiculo;  
    ...  
}
```

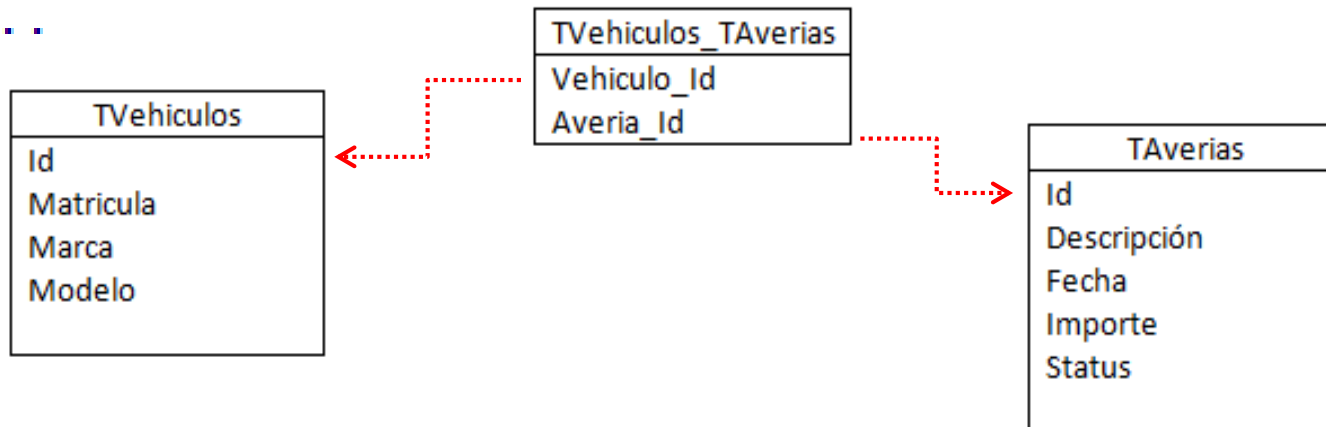


Unidireccional uno a muchos

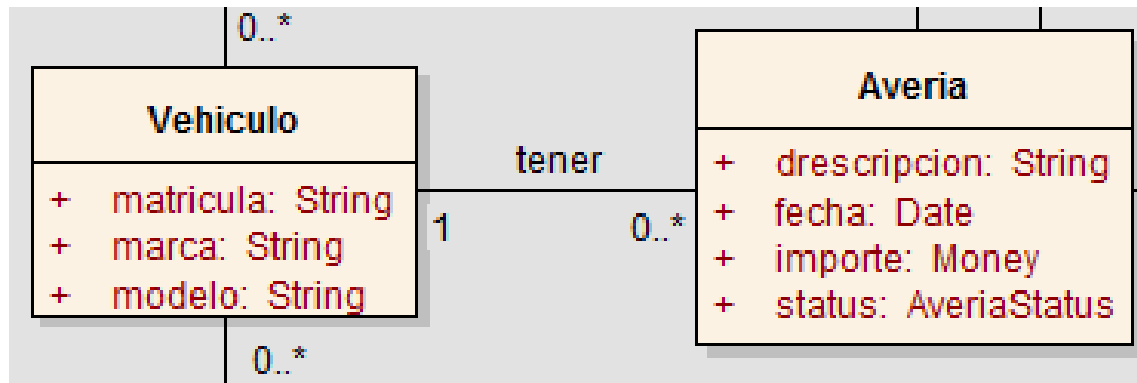


```
public class Vehiculo {  
    ...  
    @OneToMany  
    private Set<Averia> averias = new HashSet<Averia>();  
    ...  
}
```

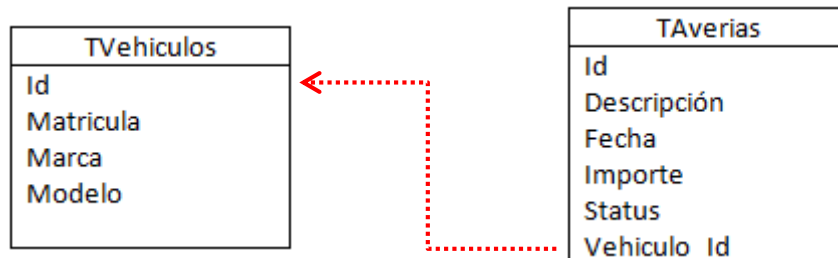
```
public class Averia {  
    ...  
}
```



Bidireccional uno a muchos



```
public class Vehiculo {  
    ...  
    @OneToMany(mappedBy = "vehiculo")  
    private Set<Averia> averias = new HashSet<Averia>();  
    ...  
}
```

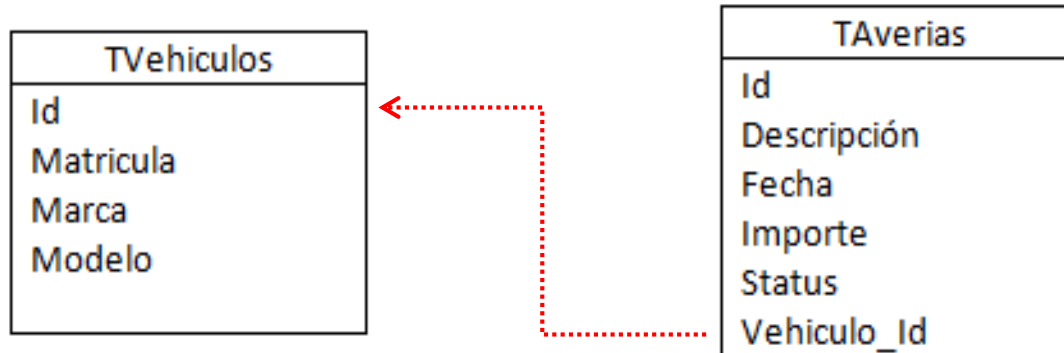


```
public class Averia {  
    ...  
    @ManyToOne  
    private Vehiculo vehiculo;  
    ...  
}
```

Vinculando los dos extremos

Con mappedBy se vinculan los extremos que son de la misma asociación

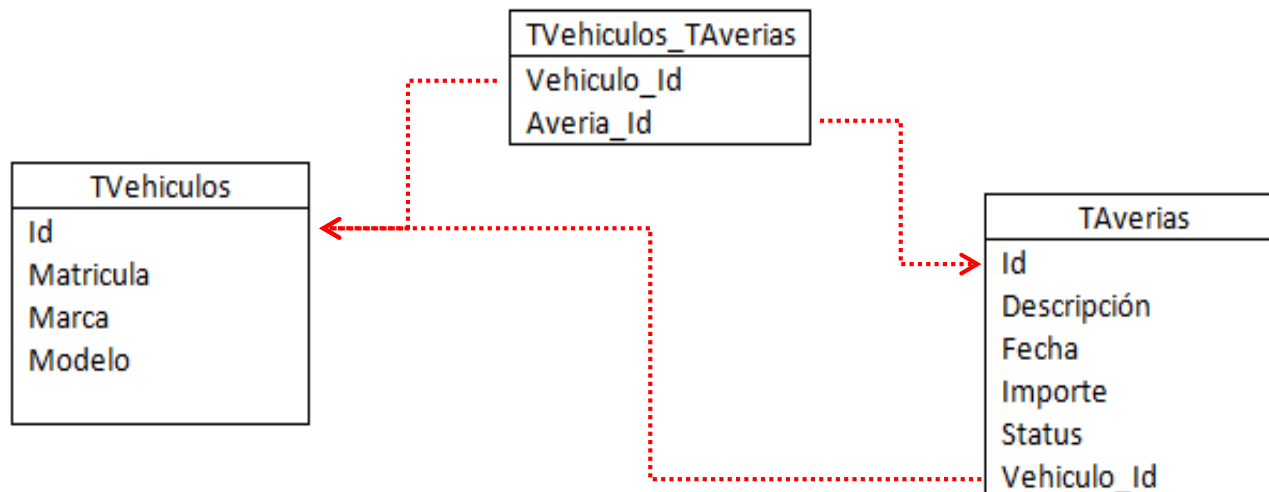
```
public class Vehiculo {  
    ...  
    @OneToMany(mappedBy = "vehiculo")  
    private Set<Averia> averias = new HashSet<Averia>();  
    ...  
}
```



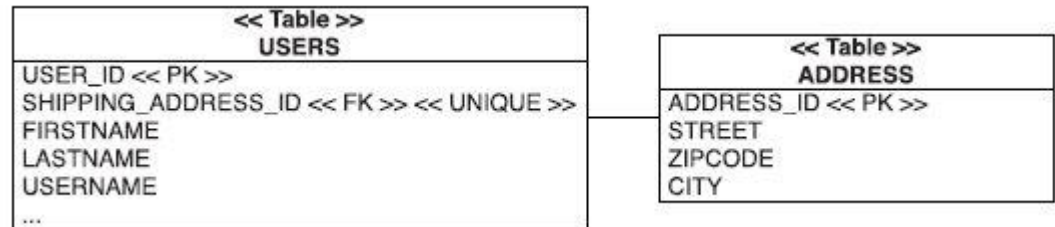
Si no se indica mappedBy se interpreta como dos asociaciones unidireccionales separadas

```
public class Vehiculo {  
    ...  
    @OneToMany  
    private Set<Averia> averias = new HashSet<Averia>();  
    ...  
}
```

```
public class Averia {  
    ...  
    @ManyToOne  
    private Vehiculo vehiculo;  
    ...  
}
```



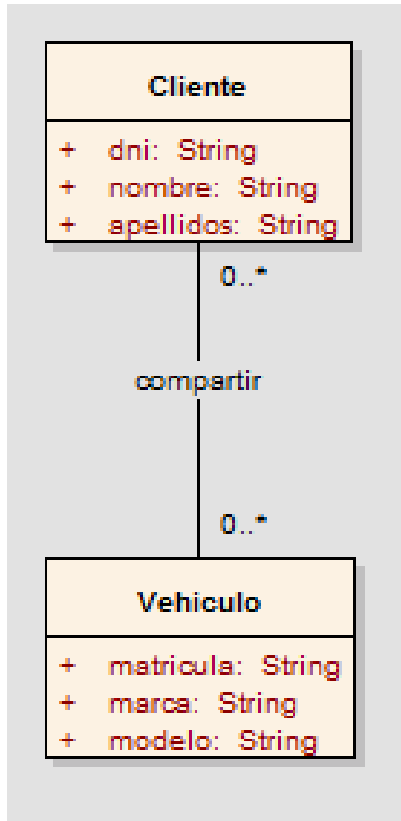
Uno o uno con foreign key



```
public class User {  
    ...  
    @OneToOne  
    @JoinColumn(name="SHIPPING_ADDRESS_ID")  
    private Address shippingAddress;  
    ...  
}  
  
public class Address {  
    ...  
    @OneToOne(mappedBy = "shippingAddress")  
    private User user;  
    ...  
}
```

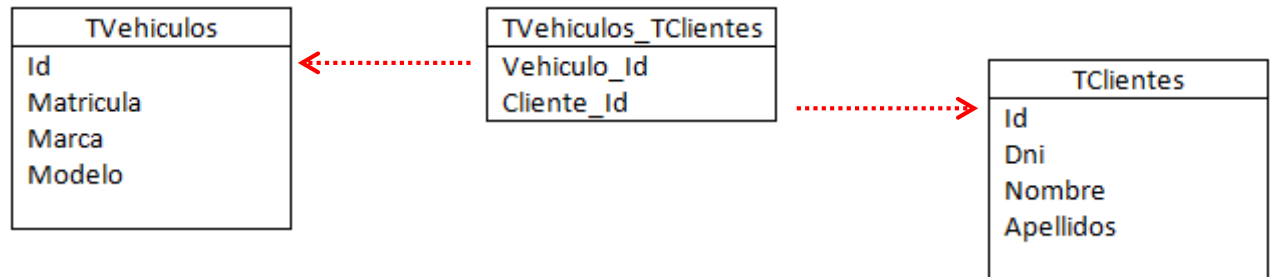
En la clase que no
tiene la FK

Muchos a muchos bidireccional



```
public class Cliente {
    ...
    @ManyToMany(mappedBy="clientes")
    private Set<Vehiculo> vehiculos = new HashSet<Vehiculos>();
    ...
}

public class Vehiculo {
    ...
    @ManyToMany
    private Set<Cliente> clientes = new HashSet<Cliente>();
    ...
}
```



Propagación en cascada

... o persistencia por alcanzabilidad

... o persistencia transitiva

```
Item newItem = new Item();  
Bid newBid = new Bid();  
  
newItem.addBid(newBid); //  
  
session.save(newItem);  
session.save(newBid);
```

Si no hay cascada
hay que salvar los
dos objetos aunque
estén asociados

```
public class Item {  
    ...  
    @OneToMany(  
        cascade = { CascadeType.PERSIST, CascadeType.MERGE },  
        mappedBy = "item")  
    private Set<Bid> bids = new HashSet<Bid>();  
    ...  
}
```

```
Item newItem = new Item();  
Bid newBid = new Bid();  
  
newItem.addBid(newBid); //  
  
session.save(newItem);
```

Con cascada basta salvar
al padre (persistencia por
alcanzabilidad)

Cascada o persistencia transitiva

- Se da en las relaciones padre/hijo
 - Los hijos dependen del padre
 - Caso especial de relaciones UNO a MUCHOS
- Se puede especificar por separado el tipo cascada

```
public class Item {  
    ...  
    @OneToMany(cascade = { CascadeType.PERSIST,  
        CascadeType.MERGE,  
        CascadeType.REMOVE },  
        mappedBy = "item")  
    private Set<Bid> bids = new HashSet<Bid>();  
    ...  
}
```

En doc de referencia
buscar tipos de cascada
"Transitive persistence"

Tipos de cascada JPA

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

Cascade delete-orphan

```
// no cascade delete-orphan  
anItem.getBids().remove(aBid);  
em.remove(aBid);
```

```
// cascade delete-orphan  
anItem.getBids().remove(aBid);
```

```
@Entity  
private class Item {  
    ...  
    @OneToMany(mappedBy = "item", orphanRemoval = true)  
    private Set<Bid> bids = new HashSet<Bid>();  
    ...  
}
```



Estrategias para mapear herencia

- JPA permite 3
- Tabla única para toda la jerarquía
 - `InheritanceType.SINGLE_TABLE`
- Tabla por cada clase no abstracta
 - `InheritanceType.TABLE_PER_CLASS`
- Tabla por cada clase
 - `InheritanceType.JOINED`

Table per class hierarchy

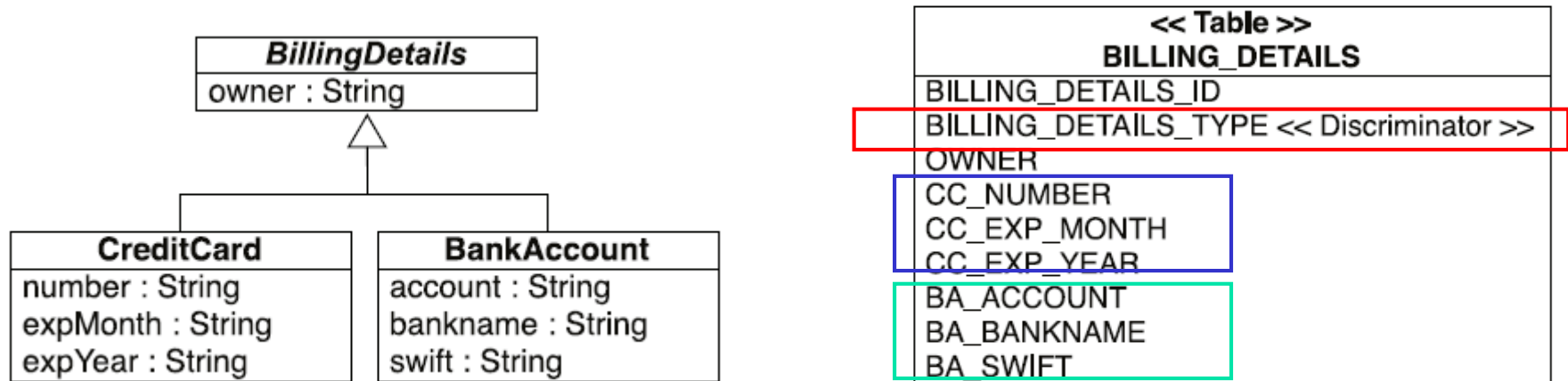


Table per class hierarchy

- Todas las clases persisten en una única tabla con la unión de todas las columnas de todas las clases
- Usa un discriminador en cada fila para distinguir el tipo
- Ventajas
 - Es simple y eficiente
 - Soporta el polimorfismo
 - Fácil de implementar
 - Fácil modificar cualquier clase
- Desventajas
 - Todas las columnas no comunes deben ser nulables
 - Las columnas nulables pueden complicar las consultas y hacer que sean más propensas a tener bugs.
 - Van a quedar columnas vacías

Table per class hierarchy

- Mapeo

- En la clase raíz añadir @DiscriminatorColumn
- En cada clase hija añadir @DiscriminatorValue

- Recomendación

- Si las clases hijas tienen pocas propiedades (se diferencian más en comportamiento) y se necesitan asociaciones polimórficas
- Debería ser tomada como estrategia por defecto

Table per class hierarchy

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name = "BILLING_DETAILS_TYPE",
    discriminatorType = DiscriminatorType.STRING
)
```

```
public abstract class BillingDetails {
    @Id @GeneratedValue
    private Long id = null;
    private String owner;
    ...
}
```

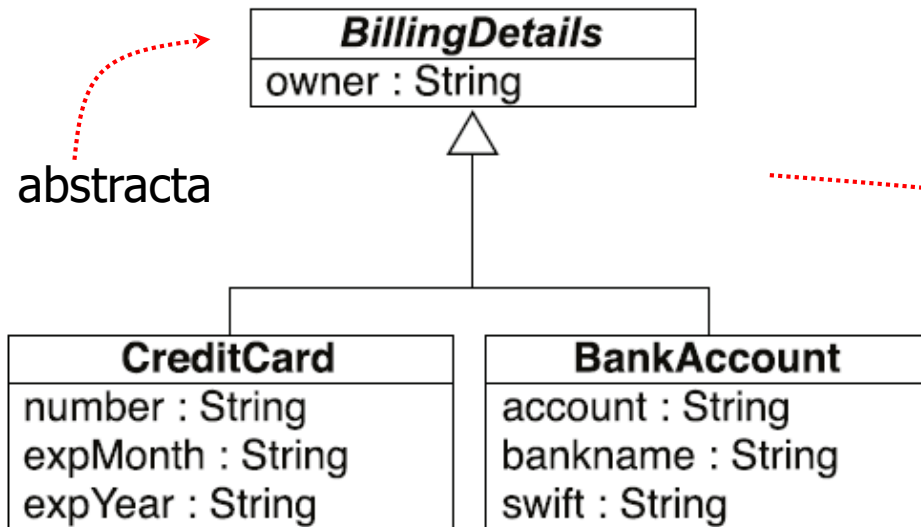
@DiscriminatorColumn,
@DiscriminatorValue
no son necesarios, se toman valores por defecto si no están presentes

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
    private String number;
    ...
}
```

Table per concrete class

- Una tabla por cada clase **no abstracta**
- Las propiedades heredadas se repiten en cada tabla
- Problemas:
 - Asociaciones polimórficas (de la superclase) se hacen poniendo la FK en cada tabla
 - Consultas polimórficas son menos eficientes, son varias SELECT o una UNION
 - Cambios en la superclase se propagan por todas las tablas
- Ventajas:
 - Cuando sólo se necesitan consultas contra las clases hijas
- Recomendable:
 - Cuando no sea necesario el polimorfismo

Table per concrete class



abstracta

Se crea una tabla por cada clase **no** abstracta

`"fom BillingDetails where owner = ?"`

`select CREDIT_CARD_ID, OWNER, 1
from CREDIT_CARD`

`select BANK_ACCOUNT_ID, OWNER,
from BANK_ACCOUNT`

<< Table >> CREDIT_CARD	
CREDIT_CARD_ID	
OWNER	
NUMBER	
EXP_MONTH	
EXP_YEAR	

<< Table >> BANK_ACCOUNT	
BANK_ACCOUNT_ID	
OWNER	
ACCOUNT	
BANKNAME	
SWIFT	

Table per concrete class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @Id @GeneratedValue
    @Column(name = "BILLING_DETAILS_ID")
    private Long id = null;
    @Column(name = "OWNER", nullable = false)
    private String owner;
    ...
}
```

Atención: Opcional en JPA, puede que no todos los proveedores JPA la soporten

```
@Entity
@Table(name = "CREDIT_CARD")
public class CreditCard extends BillingDetails {
    @Column(name = "NUMBER", nullable = false)
    private String number;
    ...
}
```

Table per subclass

- Cada clase de la jerarquía tiene su propia tabla
- Las relaciones de herencia se resuelven con FK
- Cada tabla solo tiene columnas para las propiedades no heredadas
- Ventaja:
 - Modelo relacional completamente normalizado
 - Integridad se mantiene
 - Soporta polimorfismo
 - Evoluciona bien
- Desventaja:
 - Si hay que hacer cosas a mano las consultas son más complicadas
 - Para jerarquías muy complejas el rendimiento en consultas puede ser peor, muchas joins

Table per subclass

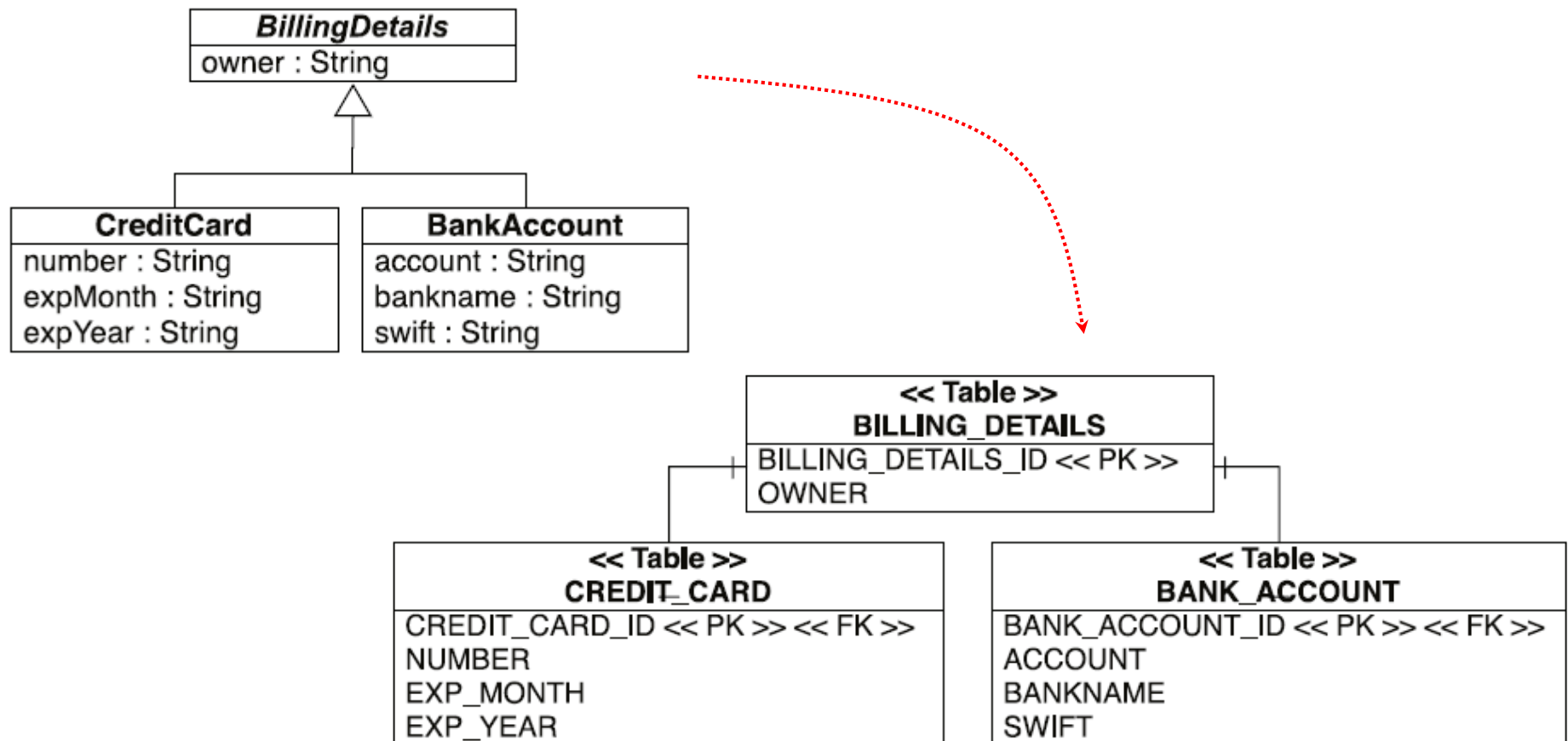


Table per subclass

■ Recomendación

- Si las clases hijas se diferencian mucho en sus propiedades y tienen muchas
- Si se necesita polimorfismo
- Cuando los nullables den problemas

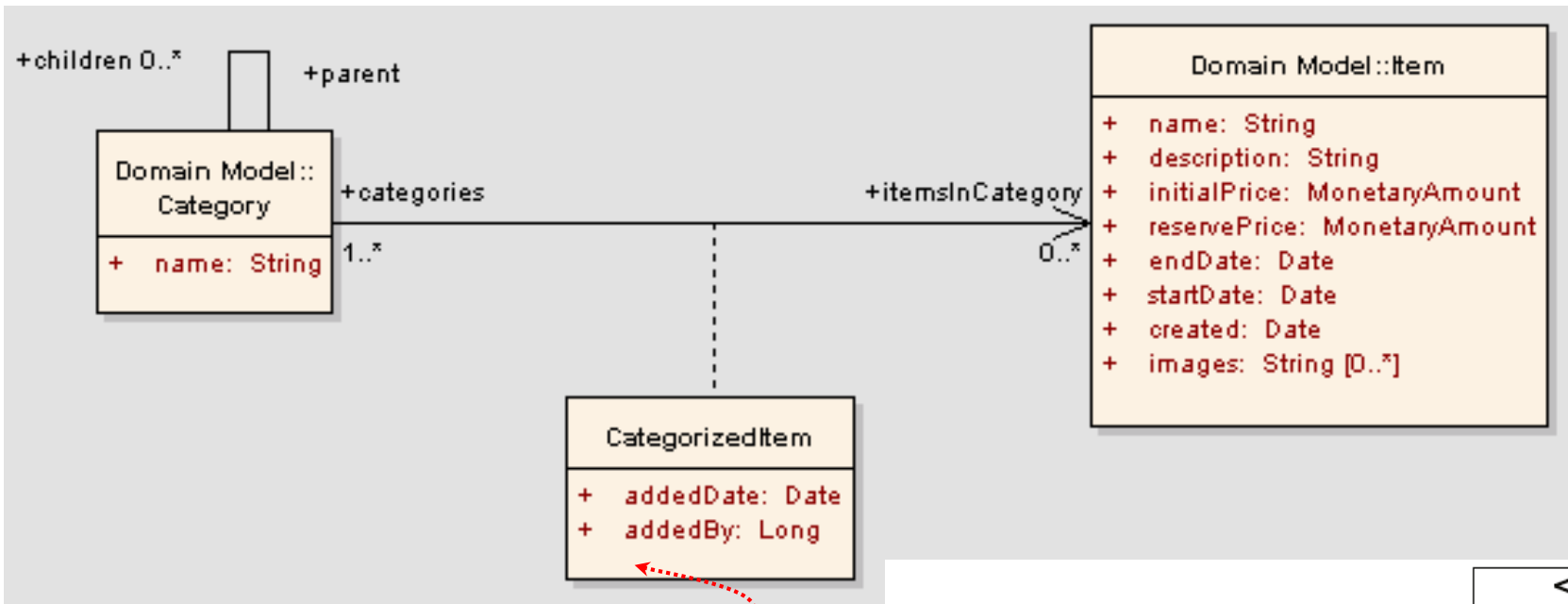
Table per subclass

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
    @Id @GeneratedValue
    private Long id = null;
    ...
}
```

```
@Entity
public class BankAccount extends BillingDetails {
    ...
}
```

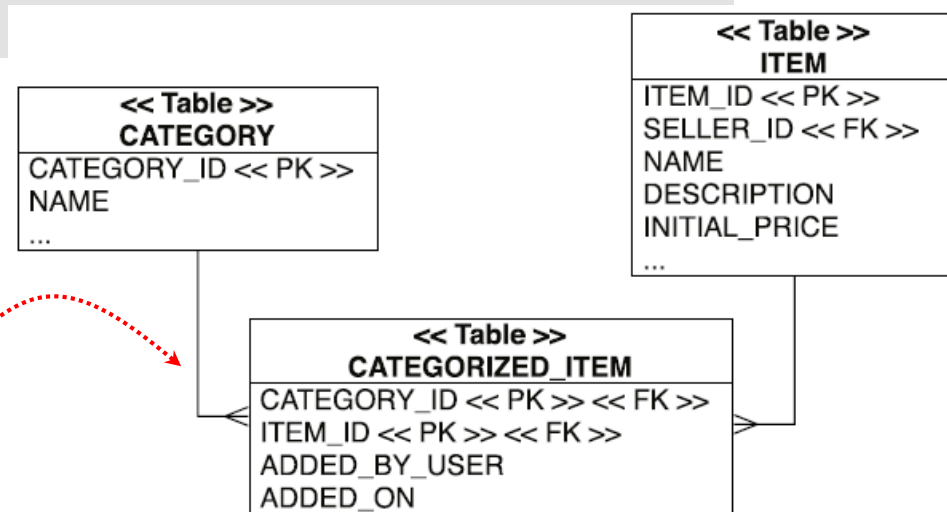


Mapeo de clases asociativas



En java es una clase más,
mapeada con dos relaciones
muchos a uno y clave
compuesta

En BDD una muchos a muchos
con más columnas



Clase asociativa

```
@Entity
@IdClass (CategorizedItemKey.class)
public class CategorizedItem {
    @Id @ManyToOne Item item;
    @Id @ManyToOne Category category;
    ...
}
```

```
public class CategorizedItemKey {
    Long item;
    Long category;
}
```

Clase para la clave compuesta

```
@Entity
public class Item {
    @Id Long id;
    ...
}
```

```
@Entity
public class Category {
    @Id Long id;
    ...
}
```



Uno a muchos con Bag

- Si no se necesita ordenación y se permiten duplicados.
 - Se usa tipo Collection en vez de Set.
- Se consigue más eficiencia.
 - Al no tener que garantizar el orden ni vigilar los duplicados, no hace falta cargar la colección para hacer las inserciones.

Uno a muchos con Bag

```
public class Bid {  
    ...  
    @ManyToOne  
    @JoinColumn(nullable = false)  
    private Item item;  
    ...  
}  
  
public class Item {  
    ...  
    @OneToMany(mappedBy = "item")  
    private Collection<Bid> bids = new ArrayList<Bid>();  
    ...  
}
```

Uno a muchos con List

- Para mantener en BDD el orden que tenían en memoria y viceversa

```
@Entity
private class Item {
    ...
    @OneToMany
    @JoinColumn(nullable = false)
    @OrderColumn
    private List<Bid> bids = new HashSet<Bid>();
    ...
}
```

```
public class Bid {
    ...
    @ManyToOne(optional=false)
    @JoinColumn(name="ITEM_ID", insertable=false, updatable=false, nullable=false)
    private Item item;
    ...
}
```

No lleva mappedBy="..."

Esto anula actualización de este extremo

Dos @JoinColumn

BID

BID_ID	ITEM_ID	BID_POSITION	AMOUNT	CREATED_ON
1	1	0	99.00	19.04.08 23:11
2	1	1	123.00	19.04.08 23:12
3	2	0	433.00	20.04.08 09:30

<...>ToMany @OrderBy

```
@Entity public class Project {  
    ...  
    @ManyToMany  
    @OrderBy("lastname ASC", "seniority DESC")  
    public List<Employee> getEmployees() {  
        ...  
    };  
    ...  
}
```

List mantiene en memoria el orden traído de BDD

pero en BDD no se mantiene el orden en el que se insertaron en List

```
@Entity public class Employee {  
    @Id  
    private int empId;  
    private String lastname;  
    private int seniority;  
    @ManyToMany(mappedBy="employees")  
    // By default, returns a List in ascending order by empId  
    private List<Project> projects;  
    ...  
}
```

Muchos a muchos unidireccional

Category
name : String

1..*

Item
name : String
description : String
initialPrice : BigDecimal
reservePrice : BigDecimal
startDate : Date
endDate : Date
state : ItemState
approvalDatetime : Date

0..*

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```

@JoinTable opcional

Se puede hacer también con List e idBag

<< Table >> CATEGORY
CATEGORY_ID << PK >>
NAME
...

<< Table >> ITEM
ITEM_ID << PK >>
SELLER_ID << FK >>
NAME
DESCRIPTION
INITIAL_PRICE
...

<< Table >> CATEGORY_ITEM
CATEGORY_ID << PK >> << FK >>
ITEM_ID << PK >> << FK >>

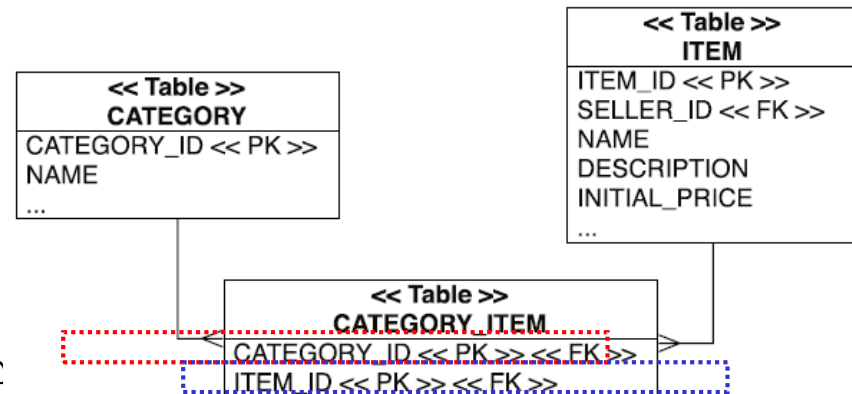
Muchos a muchos

bidireccional

```
aCategory.getItems().add(anItem);  
anItem.getCategories().add(aCategory);
```

```
@ManyToMany  
@JoinTable(  
    name = "CATEGORY_ITEM",  
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},  
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}  
)  
private Set<Item> items = new HashSet<Item>();  
  
@ManyToMany(mappedBy = "items")  
private Set<Category> categories = new HashSet<Category>();
```

@JoinTable opcional





Colecciones de Value Types

- Nuevas en JPA 2
- Sets, bags, lists, y maps de value types
- Forma estándar (idiom) de inicializar una colección

Forma de inicializar colecciones

```
private <<Interface>> images = new <<Implementation>>();
```



Siempre se declara el Interfaz genérico

Siempre se inicializan en la declaración, no en el constructor

Siempre se asigna una clase de implementación compatible con el interfaz

```
private Set<String> images = new HashSet<String>();  
...  
// Getter and setter methods
```

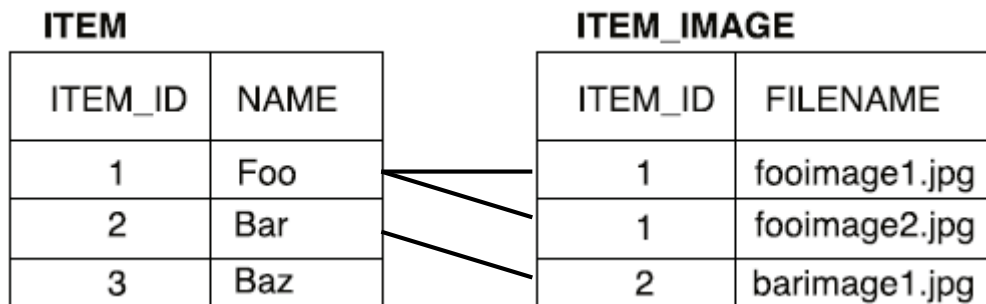

Relación entre colecciones JDK

Lo más usado
para colecciones

Interfaz	Implementación	Permite Duplicados	Preserva Orden
<code>java.util.Set</code>	<code>java.util.HashSet</code>	NO	NO
<code>java.util.SortedSet</code>	<code>java.util.TreeSet</code>	NO	SI
<code>java.util.List</code>	<code>java.util.ArrayList</code>	SI	SI
<code>java.util.Collection</code>	<code>java.util.ArrayList</code>	SI	NO
<code>java.util.Map</code>	<code>java.util.HashMap</code>	NO	NO
<code>java.util.SortedMap</code>	<code>java.util.TreeMap</code>	NO	SI
Arrays		SI	SI

Mapeo básico de Set

```
@Entity
public class Item {
    @Id @GeneratedValue
    private Long id;
    ...
    @ElementCollection
    private Set<String> images = new HashSet<String>();
    ...
}
```



La clave de ITEM_IMAGE es compuesta para evitar duplicados en el mismo ITEM (un set no los admite)

Mapeo básico de List

```
@Entity
public class Item {
    @Id @GeneratedValue
    private Long id;
    ...
    @ElementCollection
    @OrderColumn(name = "POSITION")
    private List<String> images = new ArrayList<String>();
    ...
}
```

La clave se forma con
ITEM_ID + POSITION,
se permiten duplicados
en FILENAME

nov.-17

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	POSITION	FILENAME
1	Foo	1	0	fooimage1.jpg
2	Bar	1	1	fooimage2.jpg
3	Baz	1	2	foomage3.jpg

Preserva el
orden

Mapeo básico de Map

```
@Entity
public class Item {
    @Id @GeneratedValue
    private Long id;
    ...
    @ElementCollection
    private Map<String, String> properties = new HashMap<String, String>();
    ...
}
```

Guarda las claves
del mapa

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	PROPERTIES_KEY	PROPERTIES
1	Foo	1	Image One	fooimage1.jpg
2	Bar	1	Image Two	fooimage2.jpg
3	Baz	1	Image Three	foomage3.jpg

La clave se forma
con ITEM_ID +
PROPERTIES_KEY,
no se permiten
duplicados

Mapeo de Set

```
@ElementCollection
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
private Set<String> images = new HashSet<String>();
```

@Column, @JoinTable
opcionales, solo
fuerzan nombres de
tabla y columna

ITEM		ITEM_IMAGE	
ITEM_ID	NAME	ITEM_ID	FILENAME
1	Foo	1	fooimage1.jpg
2	Bar	1	fooimage2.jpg
3	Baz	2	barimage1.jpg

La clave de ITEM_IMAGE
es compuesta para evitar
duplicados en el mismo
ITEM (un set no los
admite)

Colecciones Sorted & ordered

- El mapeador las distingue
 - **Sorted** se hace en memoria (JVM) usando **Comparable** o **Comparator**
 - **Ordered** se hace en la BBDD con SQL
- **Sorted** solo aplicable a SortedMap y SortedSet

```
private SortedMap images = new TreeMap();  
private SortedSet images = new TreeSet();
```

Sorted collections

```
@Entity
public class Item {
    @Id @GeneratedValue
    private Long id;
    ...
    @ElementCollection
    @Sort(type=SortType.NATURAL)
    private SortedSet<String> names = new TreeSet<String>();
    ...
}
```

Solo para Set y Map
(se hace en JVM)

Ordered collections

```
@Entity
public class Item {
    @Id @GeneratedValue
    private Long id;
    ...
    @ElementCollection
    @OrderBy("images desc")
    private Set<String> images = new HashSet<String>();
    ...
}
```

Para cualquier colección (excepto List()); se hace en la BDD con un **order by**