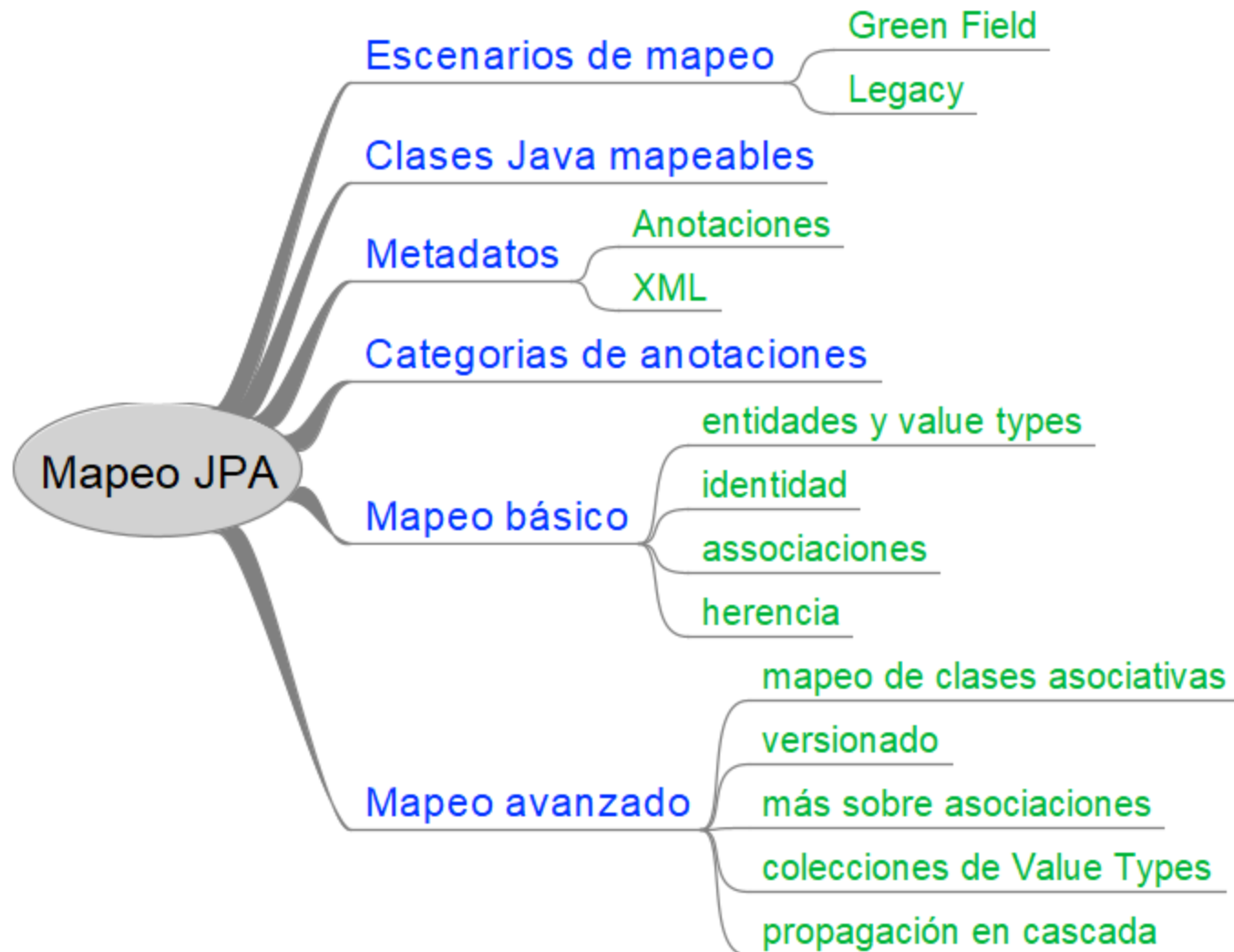


Mapeo de clases

Repositorios de Información



Escenarios de mapeo

Green Field

- El proyecto empieza de nuevo
- El diseño no está condicionado por nada anterior
- El mapeador genera la base de datos a su medida



Legacy

- Actualizamos o expandimos un sistema ya existente
- La base de datos ya existe de antes
- El mapeador se debe adaptar al diseño de una BDD ya existente y seguramente retorcido



oct.-20

alb@uniovi.es

Condiciones de una clase para ser mapeada

- Clases java planas (POJO)
- Constructor sin parámetros
 - Setters/getters etc., como siempre
 - Colecciones sobre interfaces: Set<>, List<>, etc.
- La información necesaria para persistencia se añade en forma de metadatos
 - @Annotations
 - xml

```
public class Client {  
    private String dni;  
    private String name;  
    private String surname;  
    private String email;  
    private String phone;  
    private Address address;  
  
    private Set<PaymentMean> paymentMeans = new HashSet<PaymentMean>();  
    private Set<Vehicle> vehicles = new HashSet<Vehicle>();  
  
    Client() { }  
  
    public Client(String dni) {  
        super();  
        Argument.isTrue( dni != null && dni.length() > 0 );  
        this.dni = dni;  
    }  
}
```

Entity

```
@Entity
@Table(name="TClients")
public class Client {
    @Id @GeneratedValue private Long id;

    @Column(unique=true) private String dni;
    @Basic(optional = false) private String name;
    @Basic(optional = false) private String surname;
    private String email;
    private String phone;
    private Address address;

    @OneToMany(mappedBy = "client")
    private Set<PaymentMean> paymentMeans = new HashSet<PaymentMean>();

    @OneToMany(mappedBy = "client")
    private Set<Vehicle> vehicles = new HashSet<Vehicle>();

    Client() { }

    public Client(String dni) {
        super();
        Argument.isTrue( dni != null && dni.length() > 0 );
        this.dni = dni;
    }
}
```

@Embeddable

```
public class Address {
```

```
    private String street;  
    private String city;  
    private String zipCode;
```

```
    Address() {}
```

```
    public Address(String street, String city, String zipCode) {  
        super();  
        this.street = street;  
        this.city = city;  
        this.zipCode = zipCode;  
    }
```

Value Type

Posición de @Id

Acceso por getters/setters

Acceso por atributos

```
@Entity
public class Averia {

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

Código ejecutado por el mapeador para cargar la clase en memoria

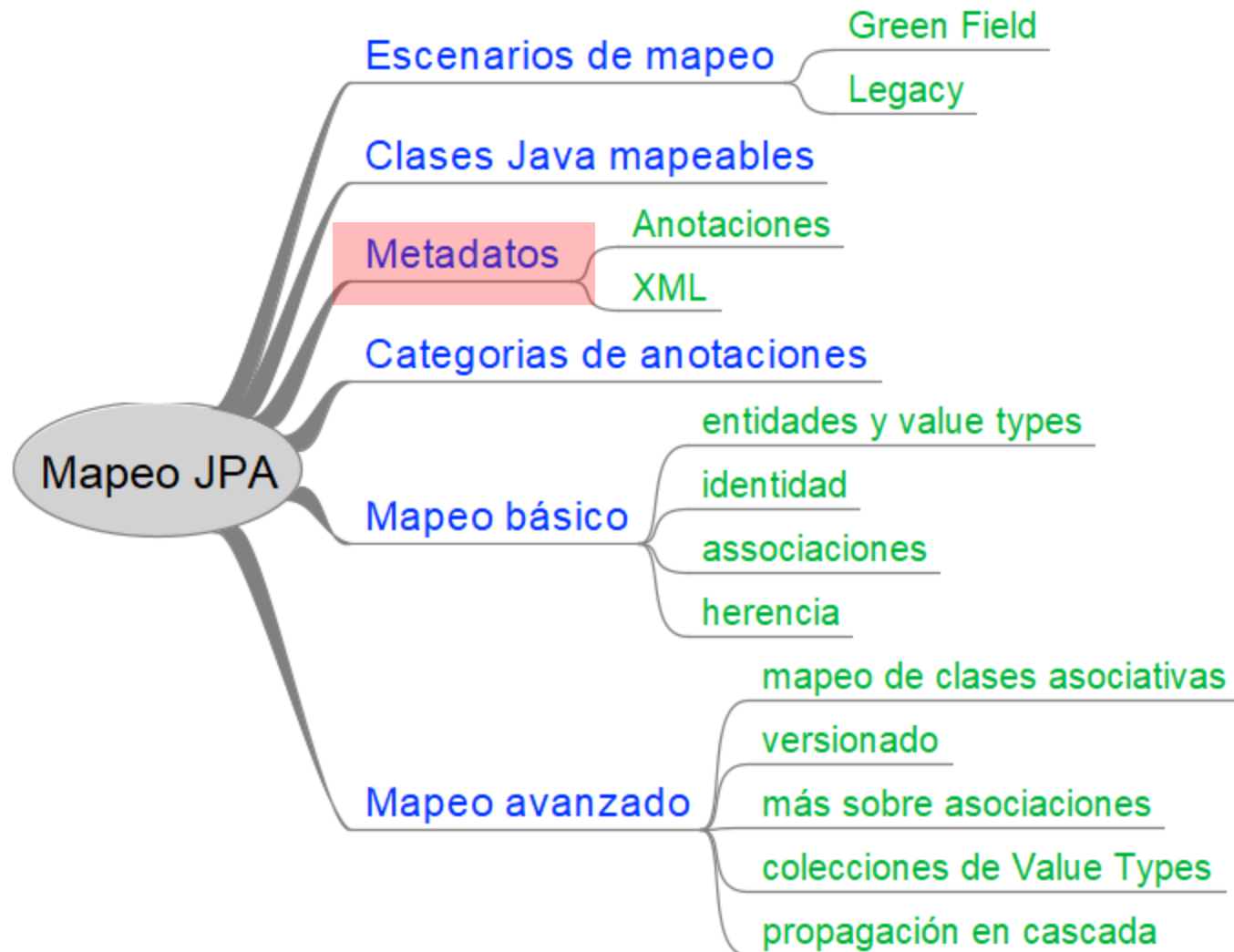
```
Averia a = new Averia();
a.id = rs.getLong("id");
...
```

```
@Entity
public class Averia {

    private Long id;
    ...

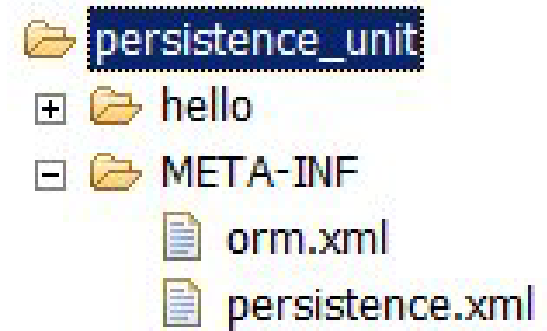
    @Id public Long getId() {
        return id;
    }
    ...
}
```

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

Metadatos en annotations

- Añadidas en Java 5
 - @Entity, @Embeddable, @Id, etc.
 - Se añaden en la clase a mapear
- Muy cómodas para el programador
 - Se compilan, detección temprana de errores



Metadatos en XML

- En fichero **orm.xml**
 - Fichero referenciado desde persistence.xml
- XML revoca las Annotations
 - En despliegue pueden se pueden ajustar rendimientos sin tocar código fuente

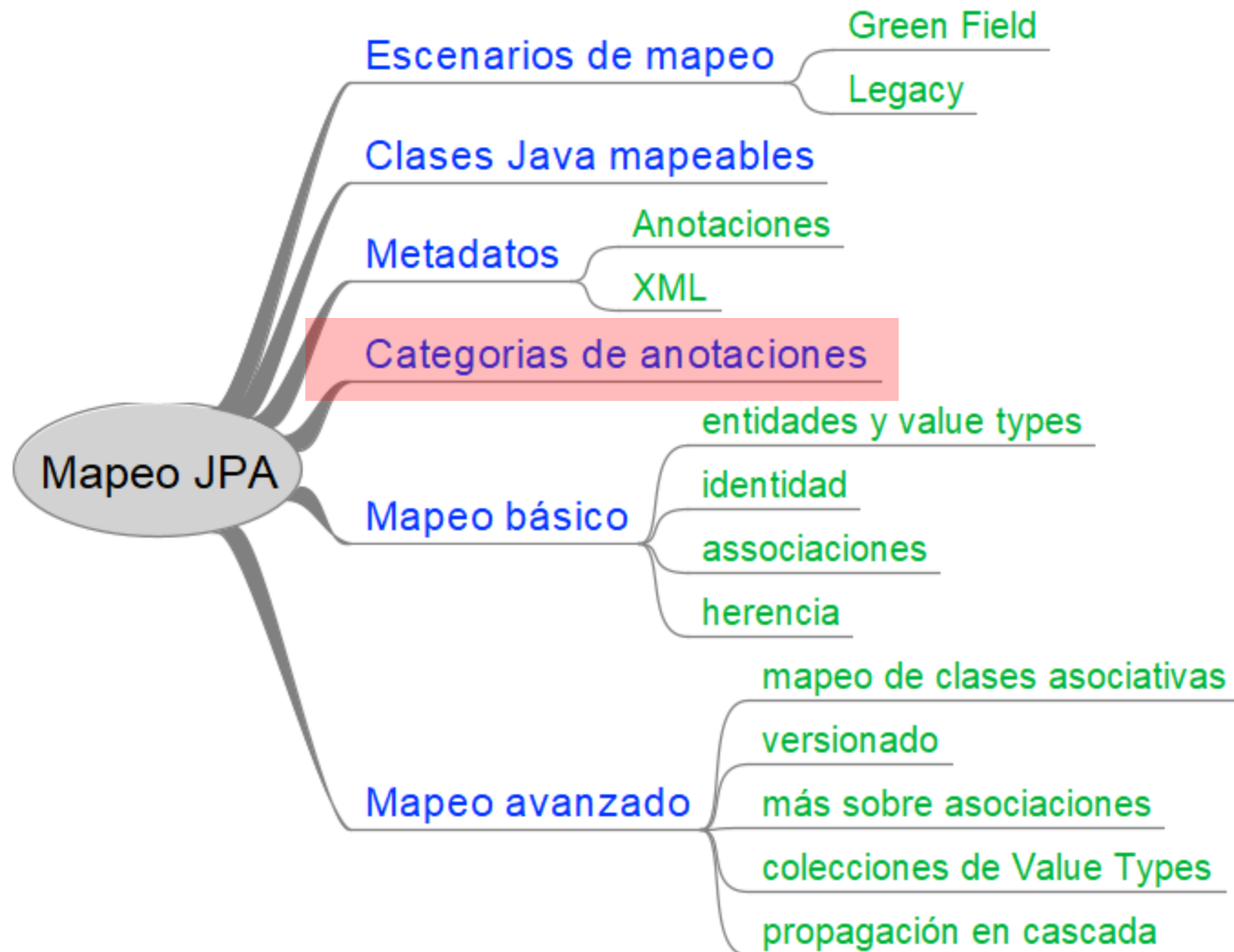
Metadatos XML, ejemplo

```
<entity class="uo.ri.cws.domain.Client">
  <table name="TClients"/>
  <attributes>
    <basic name="dni" optional="false">
      <column unique="true"/>
    </basic>
    <basic name="name" optional="false"/>
    <basic name="surname" optional="false"/>

    <one-to-many name="paymentMeans" mapped-by="client" />
    <one-to-many name="vehicles" mapped-by="client" />
  </attributes>
</entity>
```

Anotaciones vs XML

Anotaciones	XML
Cómodo programar	No tanto
Para cambiar cosas hay que recompilar	Se pueden hacer ajustes en despliegue
Ensucia el código	Deja el código limpio
Concreto, compacto	Más verboso
Detecta errores al compilar	En runtime
Mapeo disperso	Mapeo centralizado



Categorías de anotaciones

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

Anotaciones por categoría

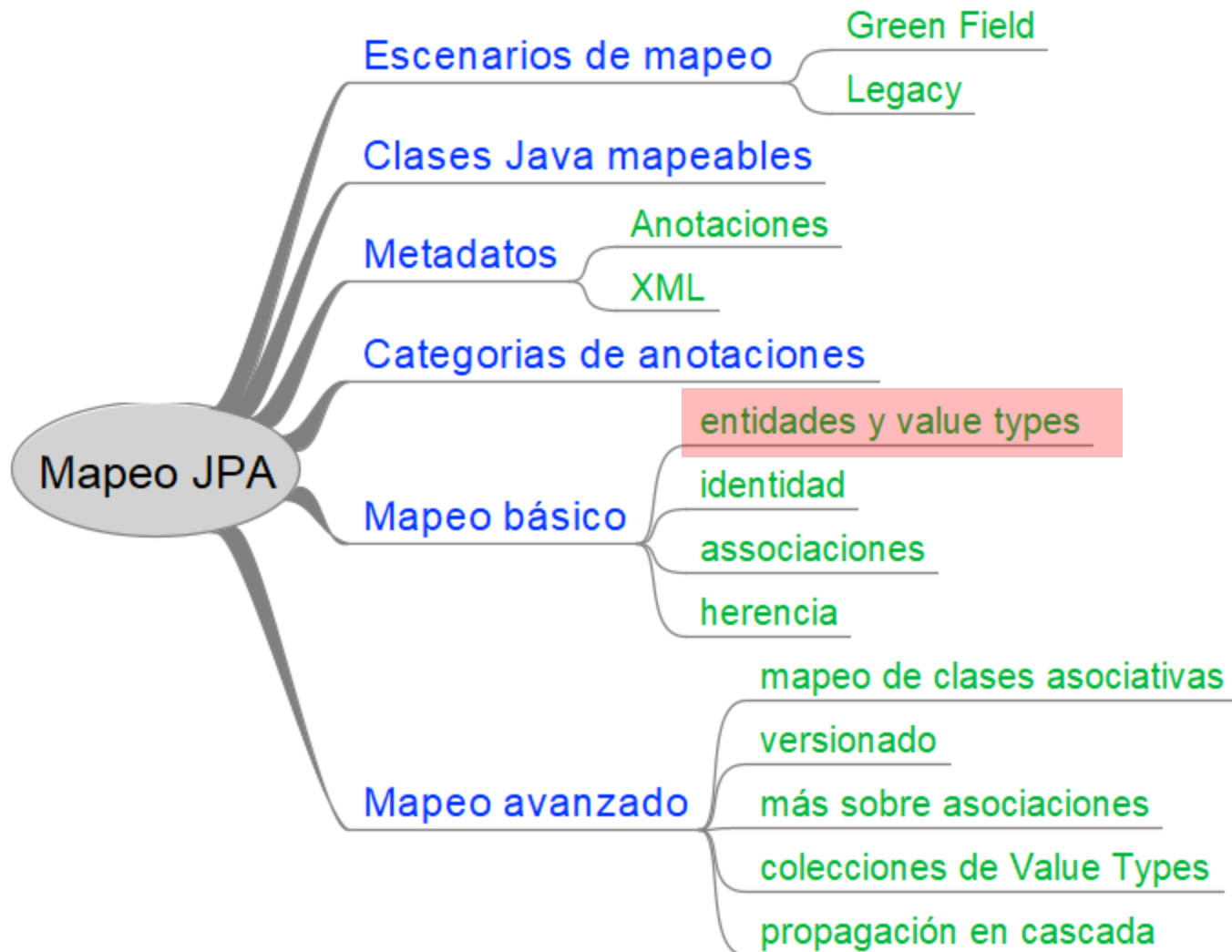
Category	Annotations
Entity	<u>@Entity</u> <u>@AccessType</u>
Database Schema Attributes	<u>@Table</u> <u>@SecondaryTable</u> <u>@SecondaryTables</u> <u>@Column</u> <u>@JoinColumn</u> <u>@JoinColumns</u> <u>@PrimaryKeyJoinColumn</u> <u>@PrimaryKeyJoinColumns</u> <u>@JoinTable</u> <u>@UniqueConstraint</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> <u>@OrderColumn</u>

Anotaciones por categoría

Category	Annotations
Composition	<u>@Embeddable</u> <u>@Embedded</u> <u>@ElementCollection</u> <u>@AttributeOverride</u> <u>@AttributeOverrides</u> <u>@AssociationOverride</u> <u>@AssociationOverrides</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>
Locking	<u>@Version</u>

Lifecycle Callback Events	<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>
Entity Manager	<u>@PersistenceUnit</u> <u>@PersistenceUnits</u> <u>@PersistenceContext</u> <u>@PersistenceContexts</u> <u>@PersistenceProperty</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
 - Tienen identidad
-
- Una entidad se mapea siempre a una tabla (o a varias en casos raros)

Entidades

@Entity

- Marca una clase como entidad

```
@Entity
public class Client {
    private String dni;
    private String name;
```

@Table (opcional)

- Indica la tabla en BBDD

```
@Entity
@Table(name="TClients")
public class Client {
    private String dni;
```

*Por defecto la
tabla se llamará
igual que la
clase*

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

Persistencia de atributos

- **Tipos Java:** Mapeo por defecto, el mapeador ya sabe cómo hacerlo
 - String, Double, Long, Date, LocalDate, Integer, Short, Boolean, Byte...
- **ValueTypes del dominio:**
 - Todos los campos a la tabla de la entidad
 - Clases `@Embeddable`, o atributos `@Embedded`
- **Extremos de asociación**
 - Foreign Key a la tabla de `@Entity`
- **Resto de casos, serialización**
 - La clase del atributo debe implementar `Serializable`

@Column

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

```
@Entity
@Table(name="TClients")
public class Client {
    @Column(unique=true) private String dni;
    @Column(name="NAME") private String nombre;
```

@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: empty String. 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 se comportará el mapeador con respecto a un atributo de tipo Java básico
- 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 guardan 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

No se necesita para los nuevos tipos:

- *LocalDate*
- *LocalTime*
- *LocalDateTime*

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

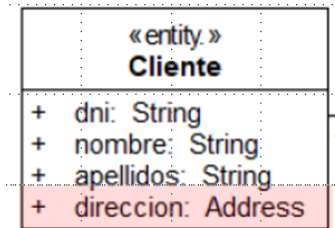
```
@Entity
@Table(name="TInvoices")
public class Invoice {

    @Column(unique=true) private Long number;
    @Temporal(TemporalType.DATE) private Date date;
```

Value Types

- Representan conceptos adicionales del dominio
- Su ciclo de vida depende de la entidad que los posee
- Semántica de composición

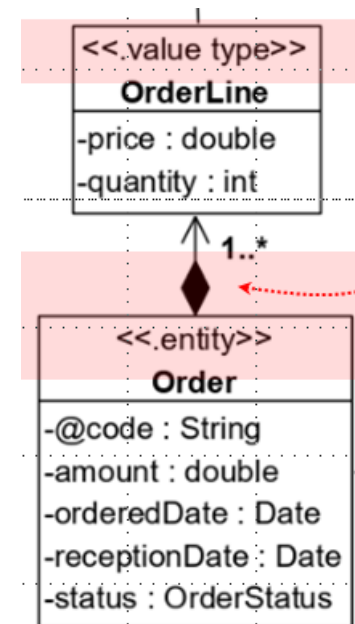
Como atributo de una entidad:



Todos los atributos del value-type serán columnas extra en la tabla de la entidad que los posee

Como colección (agregado composición UML):

Se añade tabla extra con clave ajena a la primaria de la tabla de la entidad



@Embeddable

- Marca una clase como ValueType
- Se pueden configurar los atributos con etiquetas:
 - @Basic, @Column, @Lob, @Temporal, @Enumerated

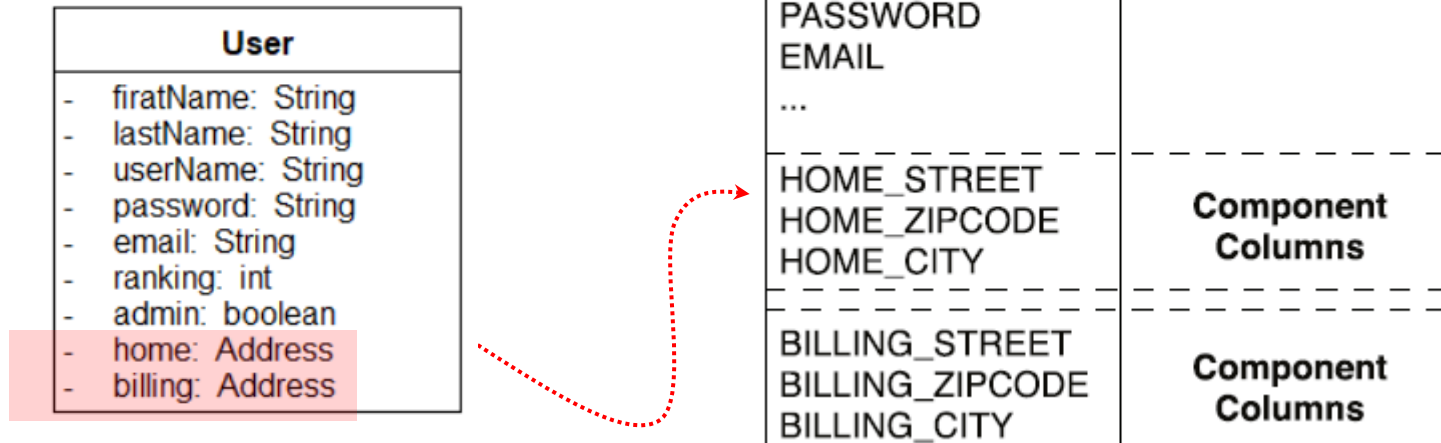
```
@Entity
@Table(name="TClients")
public class Client extends BaseEntity {
    @Column(unique=true) private String dni;
    @Basic(optional = false) private String name;
    @Basic(optional = false) private String surname;
    private String email;
    private String phone;
    private Address address;
```

```
@Embeddable
public class Address {

    private String street;
    private String city;
    private String zipCode;

    Address() {}
```

Ejemplo



```
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; }  
}
```

@ElementCollection

Señala una colección de ValueType

```
@Entity
@Table(name="TOrders")
public class Order extends BaseEntity {
```

```
//...
```

```
@ElementCollection
@CollectionTable(name="TOrderLines")
private Set<OrderLine> lines = new HashSet<>();
```



```
@Embeddable
public class OrderLine {
    @ManyToOne private SparePart sparePart;
    private double price;
    private int quantity;

    OrderLine() {}
}
```

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

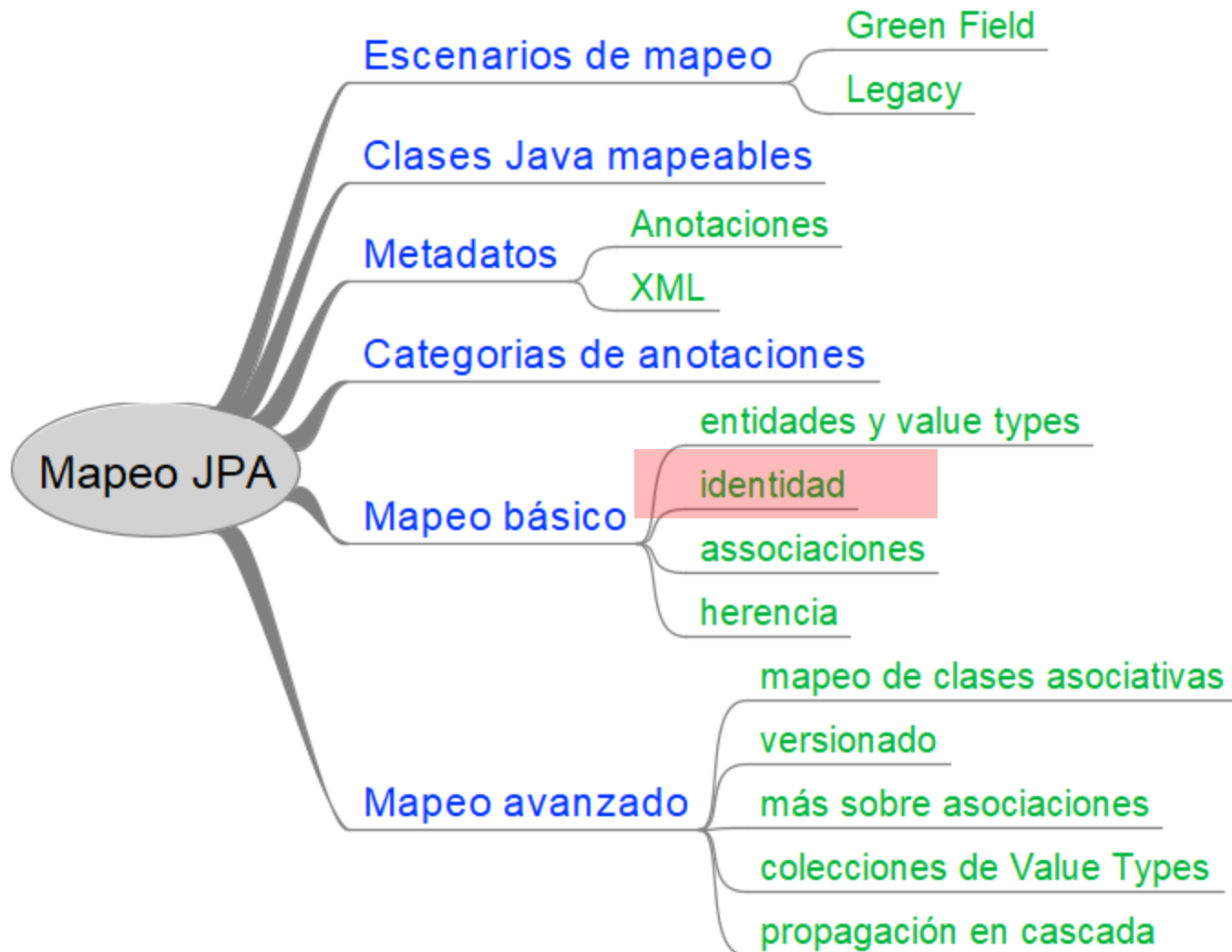
Large Object Binary: tipo de columna para almacenar tipos binarios grandes. Por ejemplo, imágenes.

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

@Transient

Marca un atributo como no persistente. Por defecto lo son todos.

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

Identity vs equality

Java identity

`a == b`

Object equality

`a.equals(b)`

Database identity

`a.getId().equals(b.getId())`

- Sobre la clave primaria de la tabla
- Se mapean con la etiqueta `@Id`
- Todas las clases **Entidad** deben tener `@Identificador`

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"

Recordatorio de BDD

Condiciones de una clave

- Nunca puede ser **NULL**
- Única para cada fila de la tabla
- Nunca puede cambiar

Tipos de claves

- Candidatas
- Naturales
- Artificiales (subrogadas)

¿Cuál es mejor para ser la primaria?

Clave candidata

- Campo (o combinaciones) que permiten determinar de forma única una fila

*Suele haber varias posibles.
Si no hay ninguna
está mal el diseño
de la tabla*

Clave natural

- Candidatas con significado para el usuario: las entiende y las maneja en el día a día (dni, nº de la SS, matrícula, etc.)

Clave artificial (subrogada)

- Sin significado en el dominio, pero sí en el sistema
- Siempre generadas por el sistema

Problema con las claves naturales...

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 ...

Estrategia recomendable

- Usar siempre claves artificiales como claves primarias
 - En Green Field, en Legacy hay que adaptarse...
- Tipo de la clave
 - Long indexa de forma eficiente
 - Pero puede ser String

*Sobre la clave natural se define un **índice único** para impedir repeticiones*

Si la clave es artificial ¿quien la genera?

Alternativas

- La introduce el usuario
- La genera la aplicación
- La genera el mapeador
- La genera la BDD

No es sencillo garantizar que no se repita nunca bajo toda circunstancia (rollbacks, concurrencia, etc...)

@Id

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

- Señala el atributo que forma clave en la tabla
- Si la clave es 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;  
    }  
}
```


Alternativas

■ ~~La introduce el usuario~~

¡Olvídalo!

■ La genera la aplicación

Vale, escribe código...

■ La genera el mapeador

@GeneratedValue

■ La genera la BDD

@GeneratedValue

- Indica que la clave no es asignada por el programa sino generada por el mapeador o por la BDD. 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		String, el nombre relaciona el generador caracterizado con @SequenceGenerator o @TableGenerator

Clave artificial, generada por la base de datos, tipo Long

```
@Entity
@Table(name="TClients")
public class Client {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY) Long id;

    @Column(unique=true) private String dni;
```

Clave artificial, generada por el mapeador

```
@Entity
@Table(name="TClients")
public class Client {
    @Id @GeneratedValue(strategy = GenerationType.TABLE) Long id;
```

Clave artificial, UUID generada por la aplicación, String

```
@Entity
@Table(name="TClients")
public class Client {
    @Id String id = UUID.randomUUID().toString();
```

@GeneratedValue

Ventajas

- Puede ser Long, indexa bien
- Descarga al programador

Inconvenientes

- Identidad inestable, no se asigna hasta FLUSH de base de datos
- hashCode() e equals() deben ir sobre la identidad natural (no sobre la generada) → identidad natural sin setters

Ventajas

- Se genera al crear entidad (constructor), identidad estable
- hashCode() e equals() sobre UUID, generalizable
- Código único y universal, permite referencias externas
- No predecible, mejora seguridad

Inconvenientes

- 36 caracteres, ocupa más
- Indexa peor que Long

@MappedSuperClass

Permite establecer opciones de mapeo comunes a varias entidades en una clase base

```
@Entity
@Table(name="TClients")
public class Client extends BaseEntity {
    @Column(unique=true) private String dni;
    @Basic(optional = false) private String name;
    @Basic(optional = false) private String surname;
    private String email;
    private String phone;
    private Address address;

    @OneToMany(mappedBy = "client")
    private Set<PaymentMean> paymentMeans = new HashSet<PaymentMean>();
}
```

@MappedSuperClass

@MappedSuperclass

```
public abstract class BaseEntity {
```

```
    @Id private String id = UUID.randomUUID().toString();
```

```
    @Version private Long version;
```

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

```
    public Long getVersion() {  
        return version;  
    }
```

```
    @Override
```

```
    public int hashCode() {
```

```
        final int prime = 31;
```

```
        int result = 1;
```

```
        result = prime * result + ((id == null) ? 0 : id.hashCode());
```

```
        return result;  
    }
```

```
    @Override
```

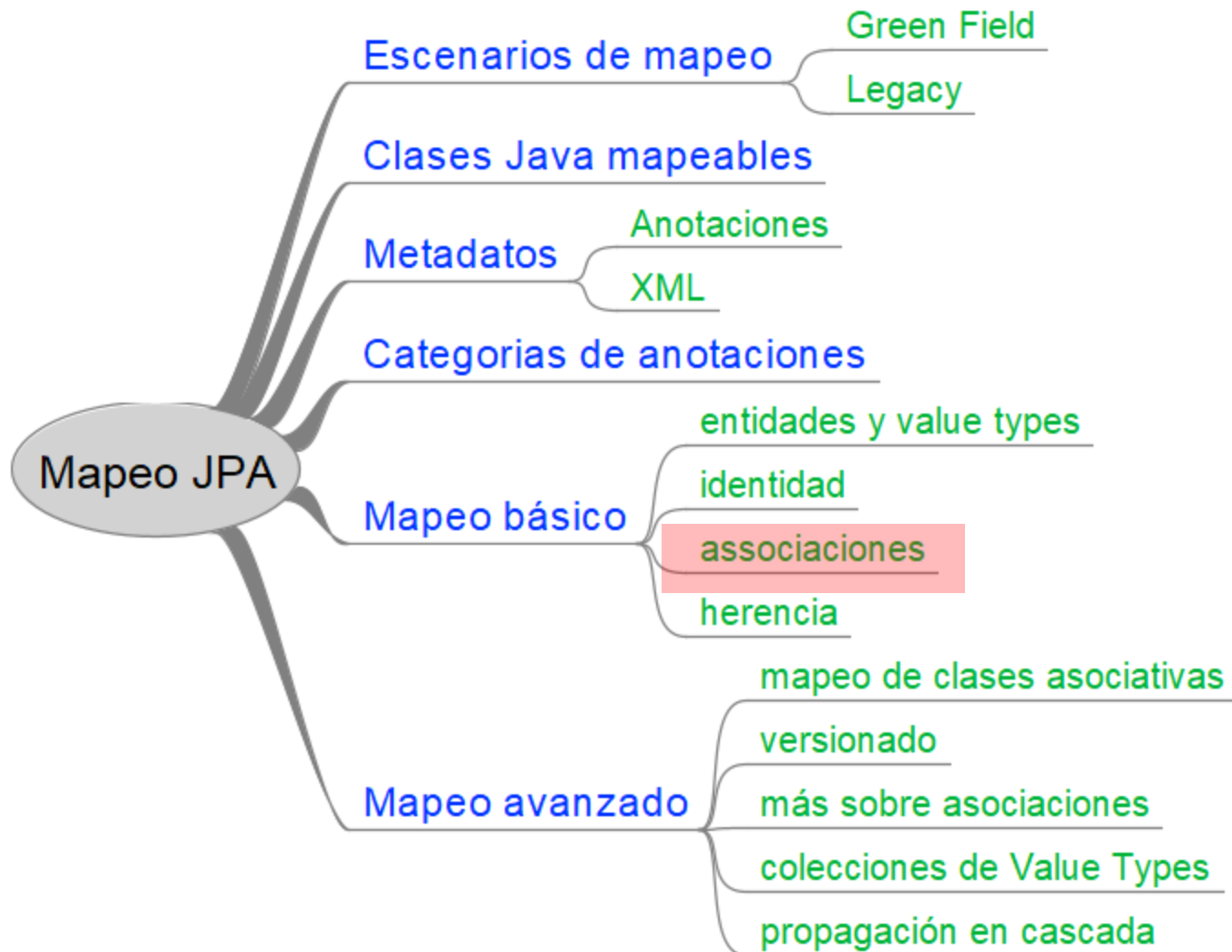
```
    public boolean equals(Object obj) {
```

```
        if (this == obj)
```

```
            return true;
```

```
        if (obj == null)
```

*hashCode()
e equals()
generalizado
sobre UUID*



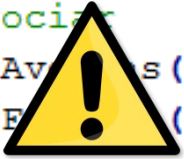
Asociaciones UML implementadas en Java

*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

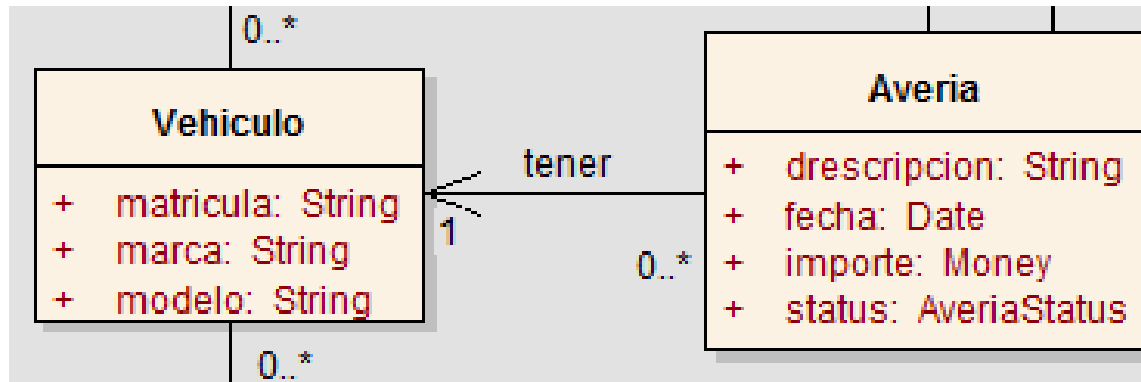
one-to-many

many-to-one



son direccionales

Unidireccional muchos a uno



```

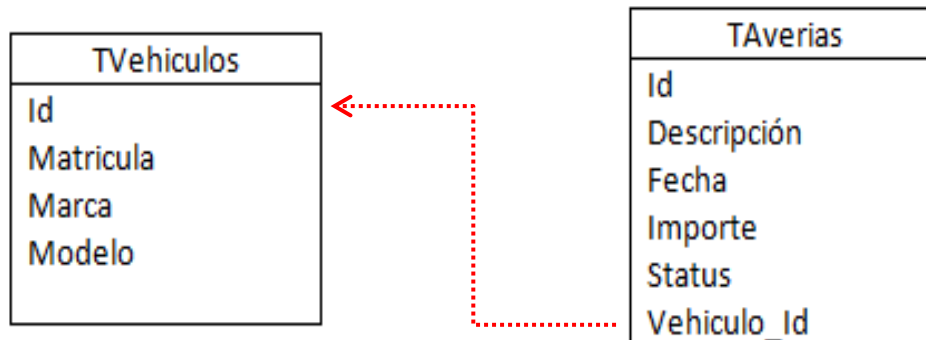
public class Vehiculo {
    ...
}

```

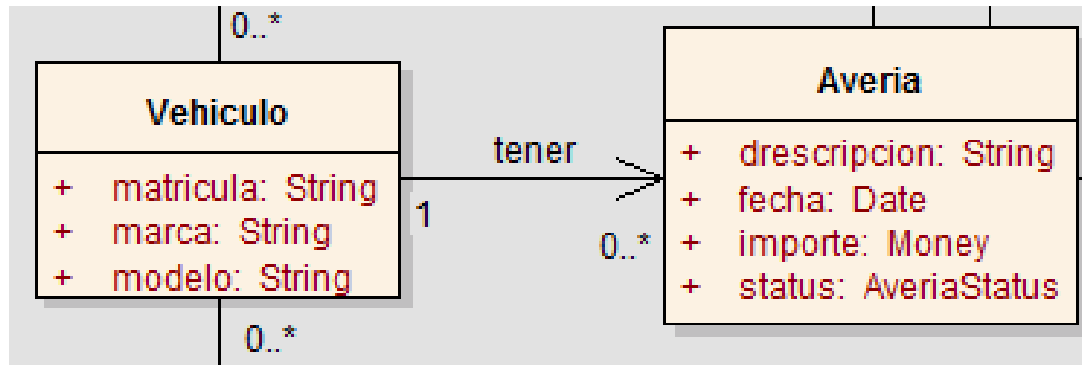
```

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

```

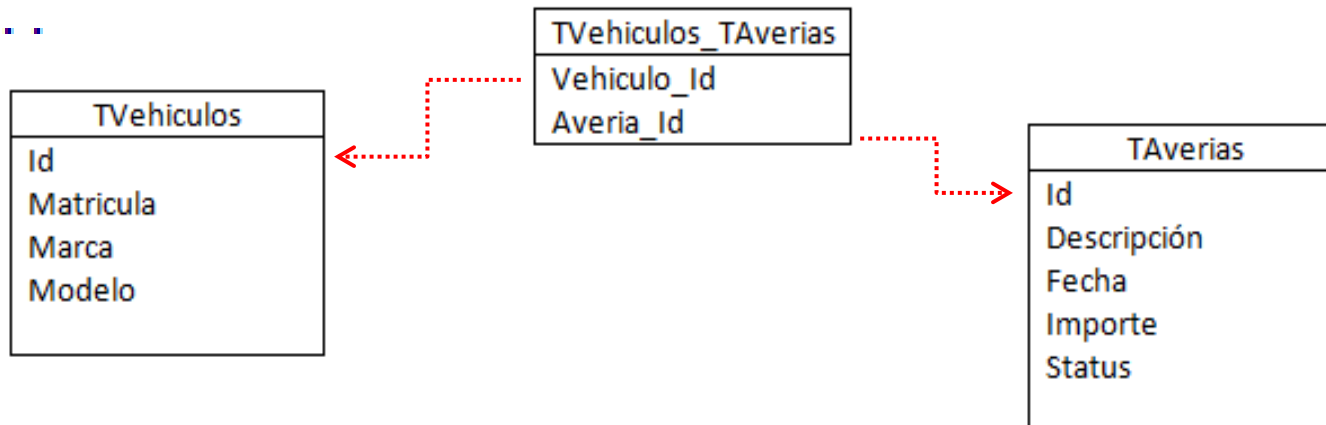


Unidireccional uno a muchos

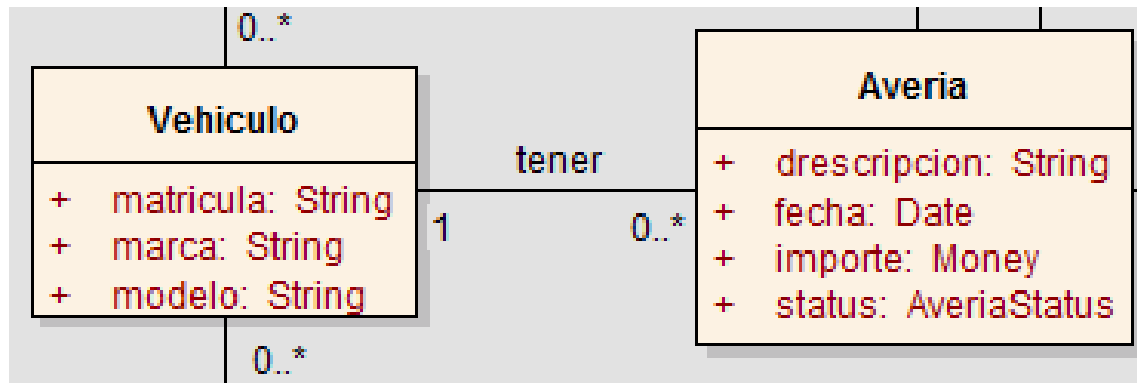


```
public class Vehiculo {  
    ...  
    @OneToMany  
    private Set<Averia> averias = new HashSet<Averia>();  
    ...  
}
```

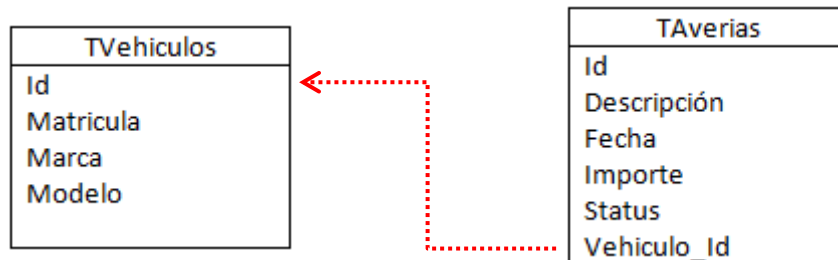
```
public class Averia {  
    ...  
}
```



Bidireccional uno a muchos



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

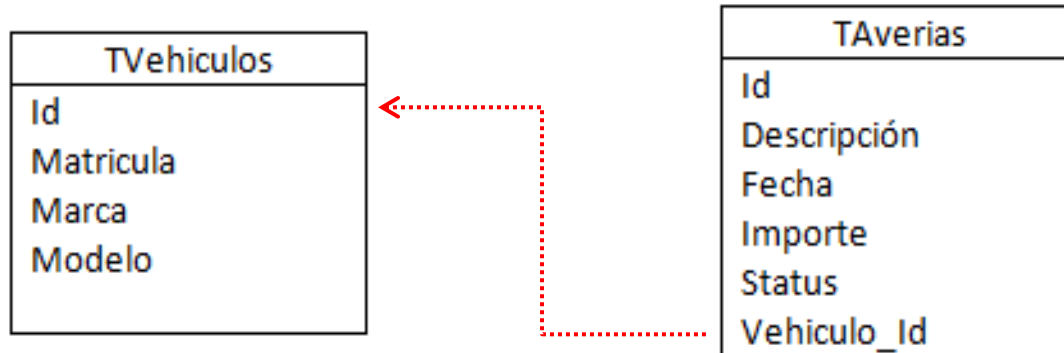


```
public class Averia {  
    ...  
    @ManyToOne  
    private Vehiculo vehiculo;  
    ...  
}
```

Vinculando los dos extremos

Con mappedBy se vinculan los extremos de la misma asociación

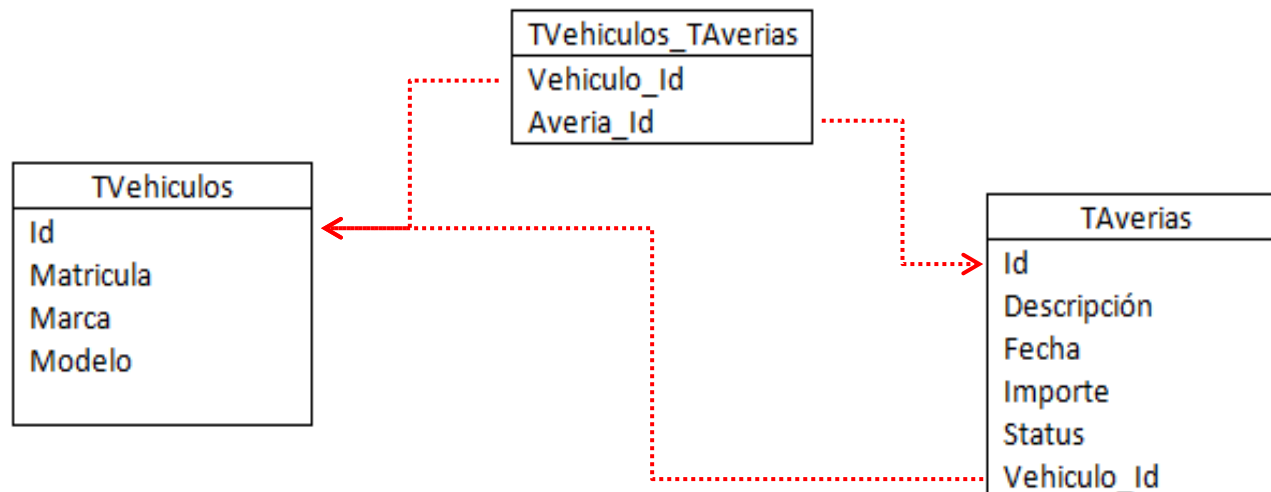
```
public class Vehiculo {  
    ...  
    @OneToMany(mappedBy = "vehiculo")  
    private Set<Averia> averias = new HashSet<Averia>();  
    ...  
}
```



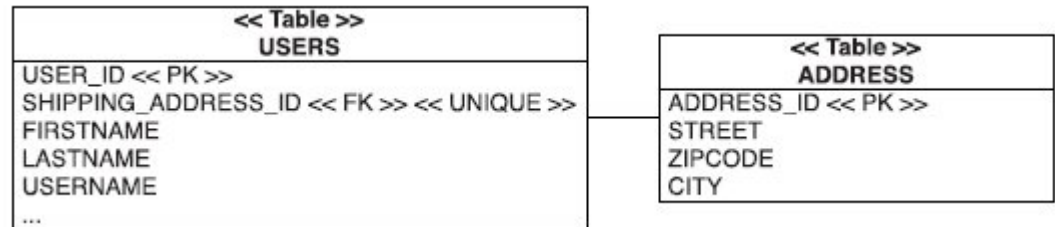
Si no se indica mappedBy se interpreta como dos asociaciones unidireccionales separadas

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

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



Uno o uno con foreign key

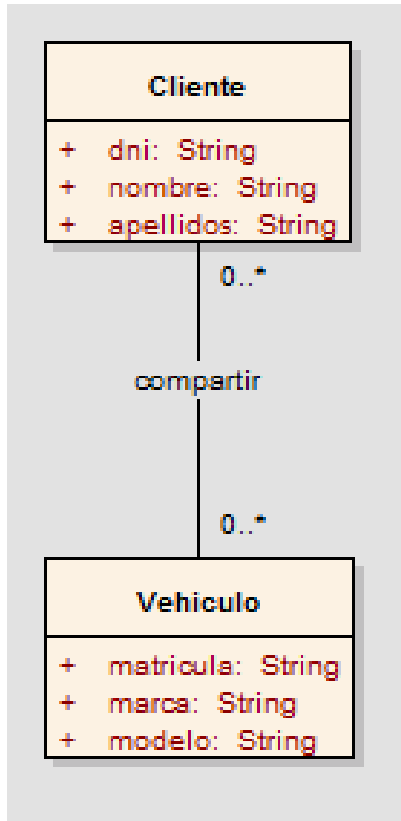


```
public class User {  
    ...  
    @OneToOne  
    @JoinColumn(name="SHIPPING_ADDRESS_ID")  
    private Address shippingAddress;  
    ...  
}
```

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

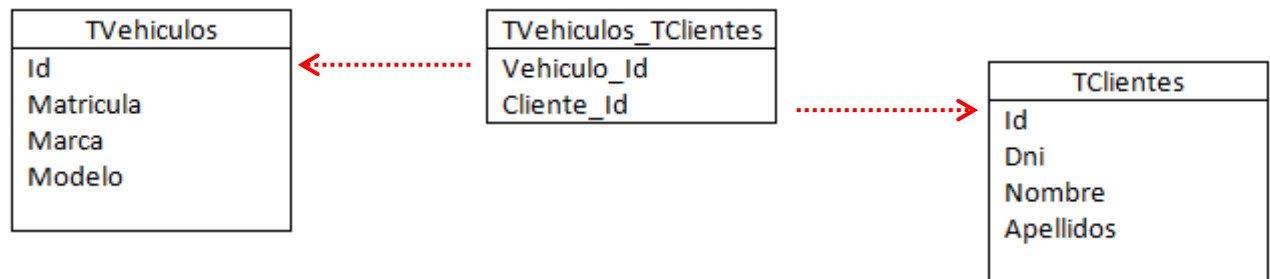
*"mappedBy" va en
la clase/tabla que
no tiene la FK*

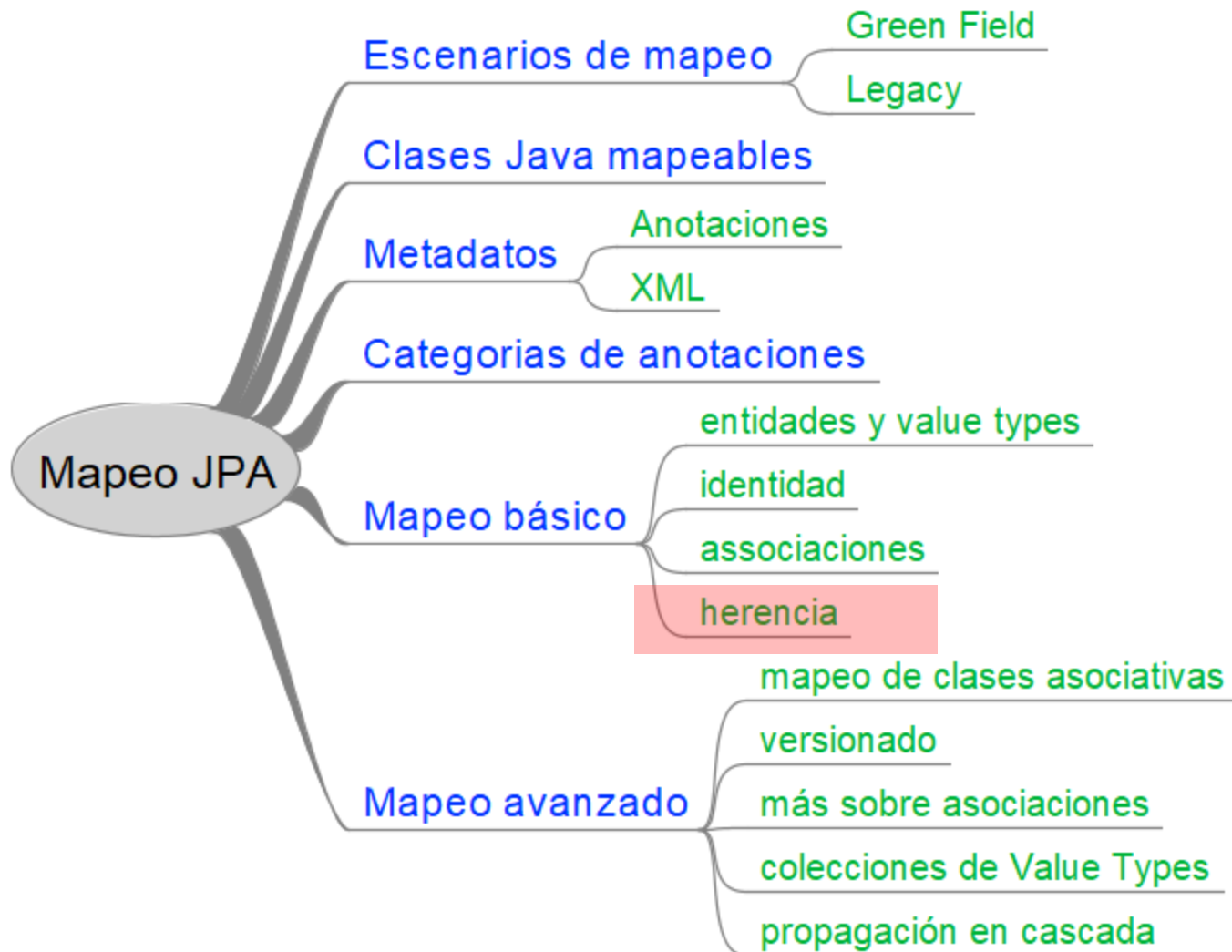
Muchos a muchos bidireccional



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

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





Estrategias para mapear herencia

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

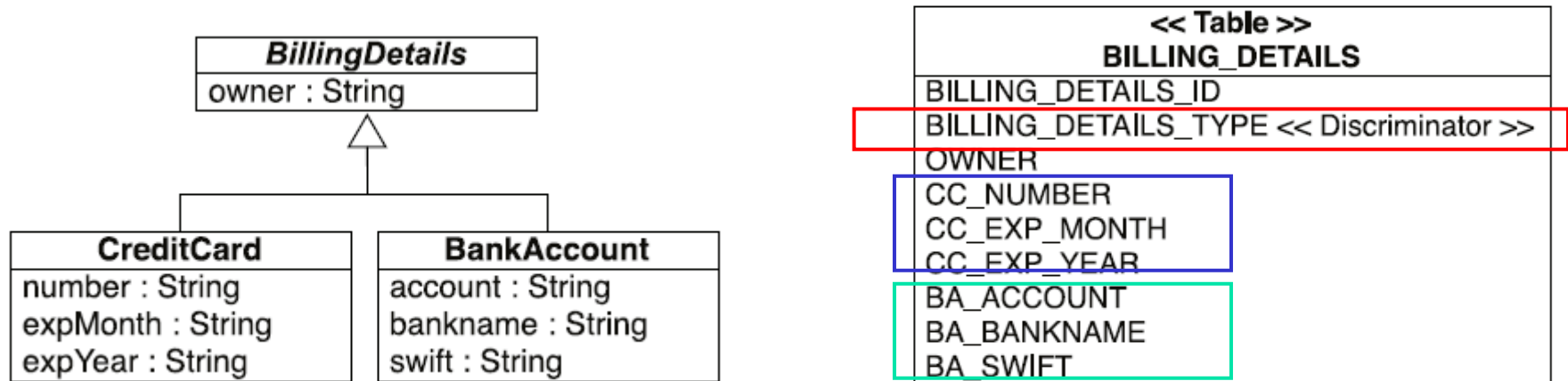


Table per class hierarchy

- 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 (desperdicio de espacio)
 - Pueden generar tablas que ocupan mucho con pocos datos

Table per class hierarchy

Estrategia por defecto del mapeador

Mapeo

- En la clase base:
 - `@Inheritance`
 - `@DiscriminatorColumn` (opcional)
- En cada clase derivada
 - `@DiscriminatorValue` (opcional)

Recomendación

- Cuando las clases derivadas añaden pocas columnas y se necesitan consultas polimórficas eficientes

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
- 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

Ventajas:

- Evita los nulos

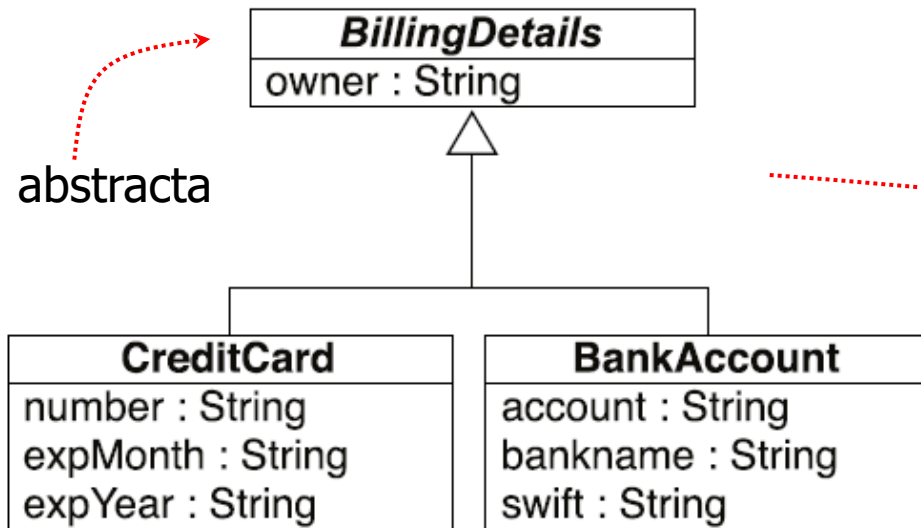
Problemas:

- 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

Recomendable:

- Cuando no sean necesarias consultas polimórficas o no críticas

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;
    ...
}
```

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

InheritanceType.JOINED

- 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

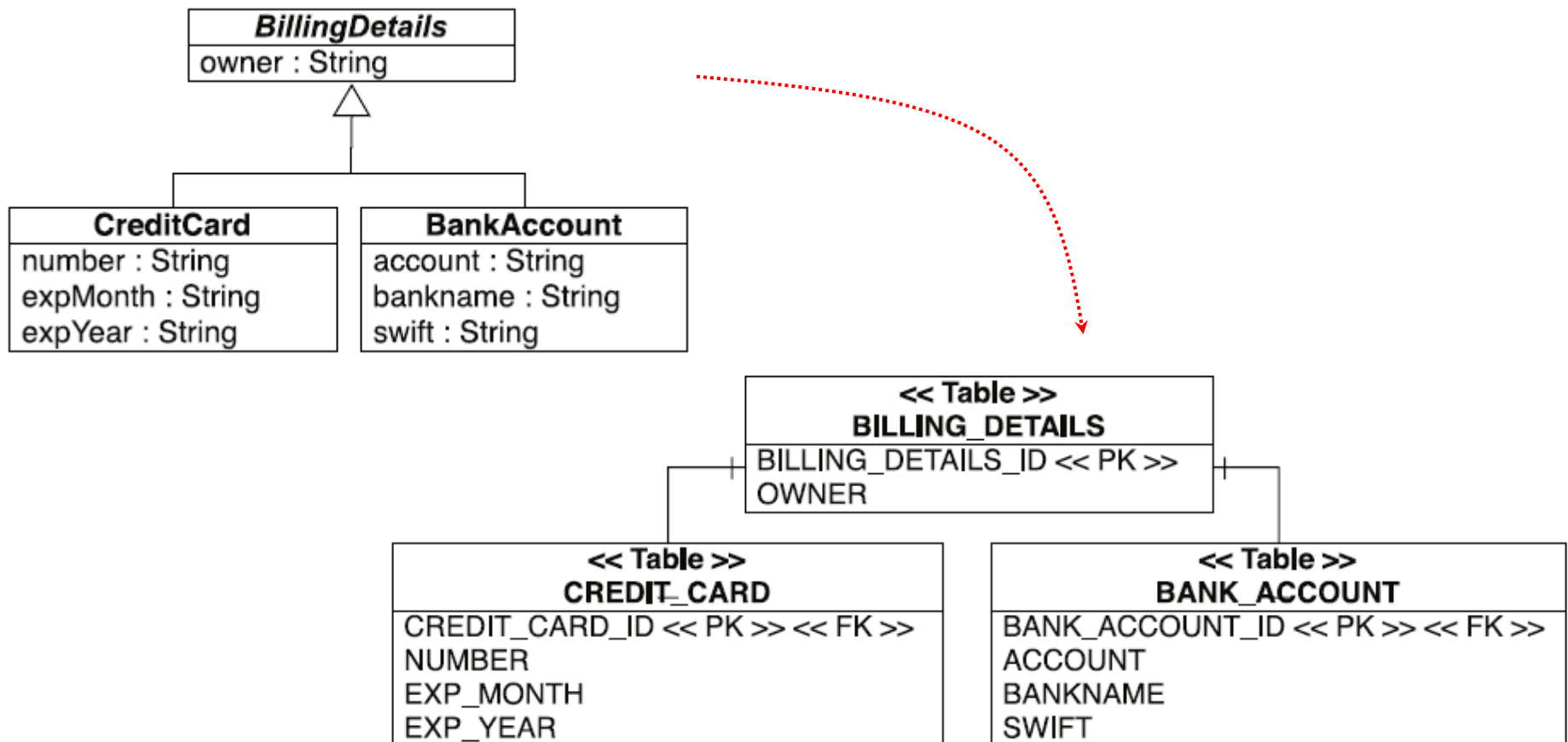


Table per subclass

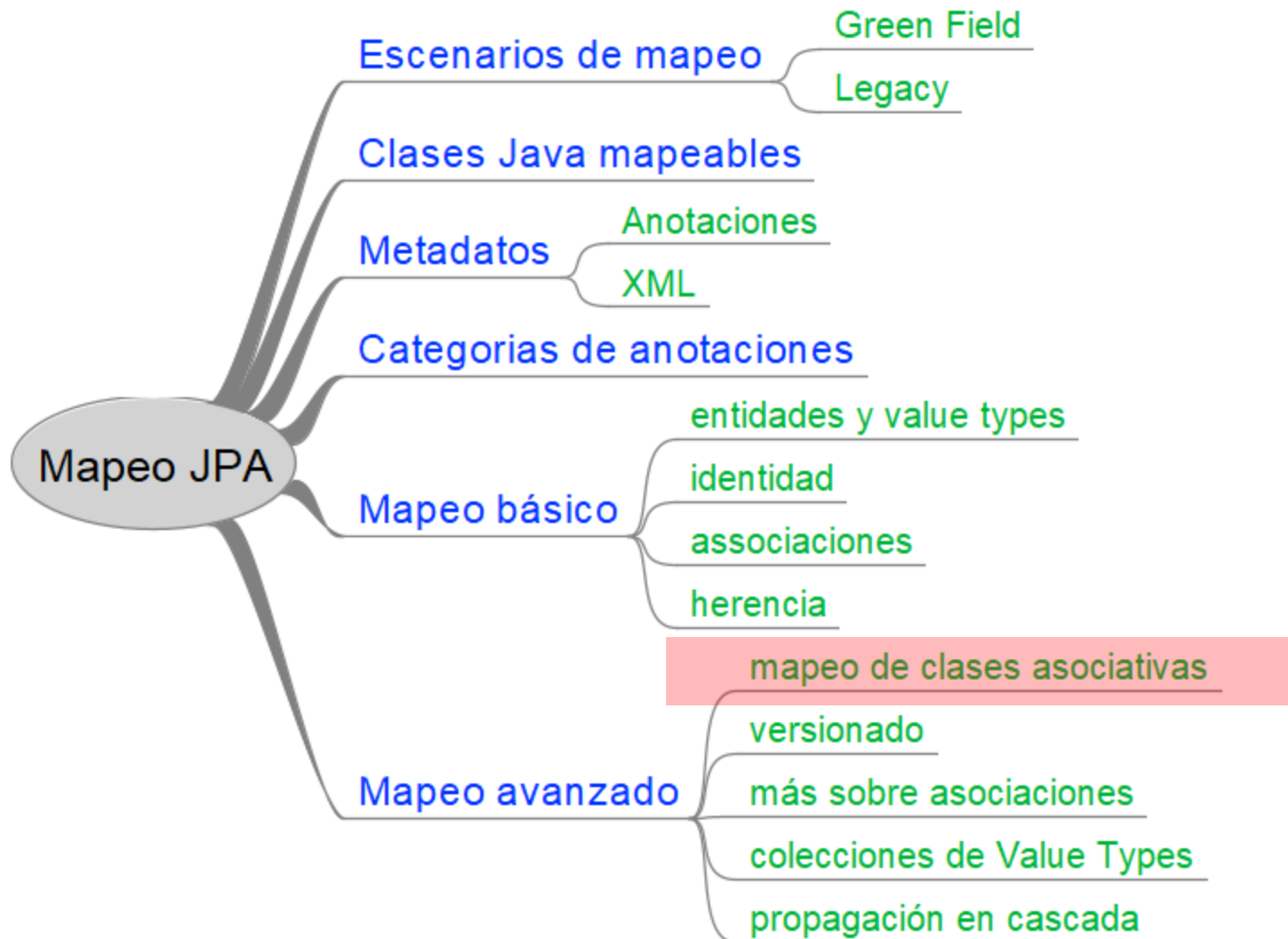
■ Recomendación

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

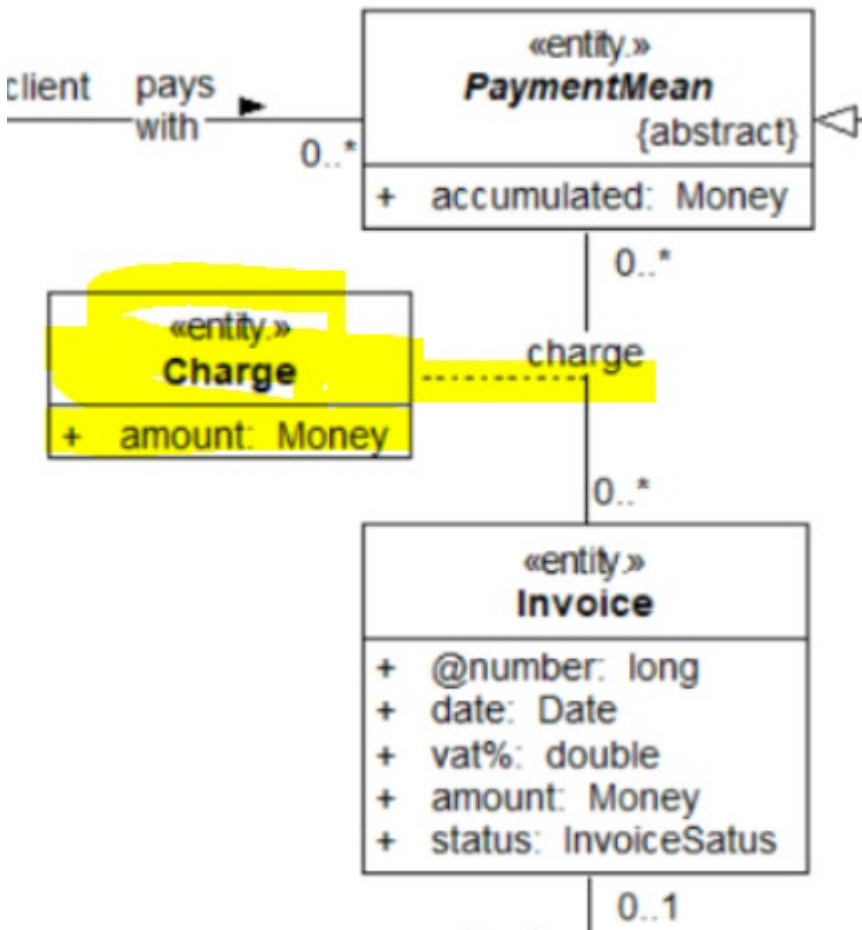
Table per subclass

```
@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,
identidad compuesta*

TCHARGES	
ID	VARCHAR(255)
AMOUNT	DOUBLE(64,0)
INVOICE_ID	VARCHAR(255)
PAYMENTMEAN_ID	VARCHAR(255)

TINVOICES	
ID	VARCHAR(255)
AMOUNT	DOUBLE(64,0)
DATE	DATE
NUMBER	BIGINT
STATUS	VARCHAR(255)
VAT	DOUBLE(64,0)

TPAYMENTMEANS	
ID	VARCHAR(255)
DTYPE	VARCHAR(31)
ACCUMULATED	DOUBLE(64,0)
CLIENT_ID	VARCHAR(255)

*En BDD tabla
con más
columnas*

Mapeo de clases asociativas

Alternativas

- Añadir identidad artificial
- Poner identidad compuesta

Con identidad artificial

```
@Entity
@Table(name="TCharges", uniqueConstraints = {
    @UniqueConstraint( columnNames = {
        "INVOICE_ID", "PAYMENT_ID"
    })
})
public class Charge {
    @Id private String id = UUID.randomUUID().toString();

    @ManyToOne private Invoice invoice;
    @ManyToOne private PaymentMean paymentMean;
    private double amount = 0.0;

    Charge() {}
}
```

Índice compuesto en la tabla

Con identidad compuesta

```
@Entity
@Table(name="TCharges",
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {
            "INVOICE_ID",
            "PAYMENTMEAN_ID"
        })
    }
)
```

Clase para la clave compuesta (metadatos)

```
@IdClass( ChargeKey.class )
```

```
public class Charge extends BaseEntity {
    @Id @ManyToOne private Invoice invoice;
    @Id @ManyToOne private PaymentMean paymentMean;
    private double amount = 0.0;

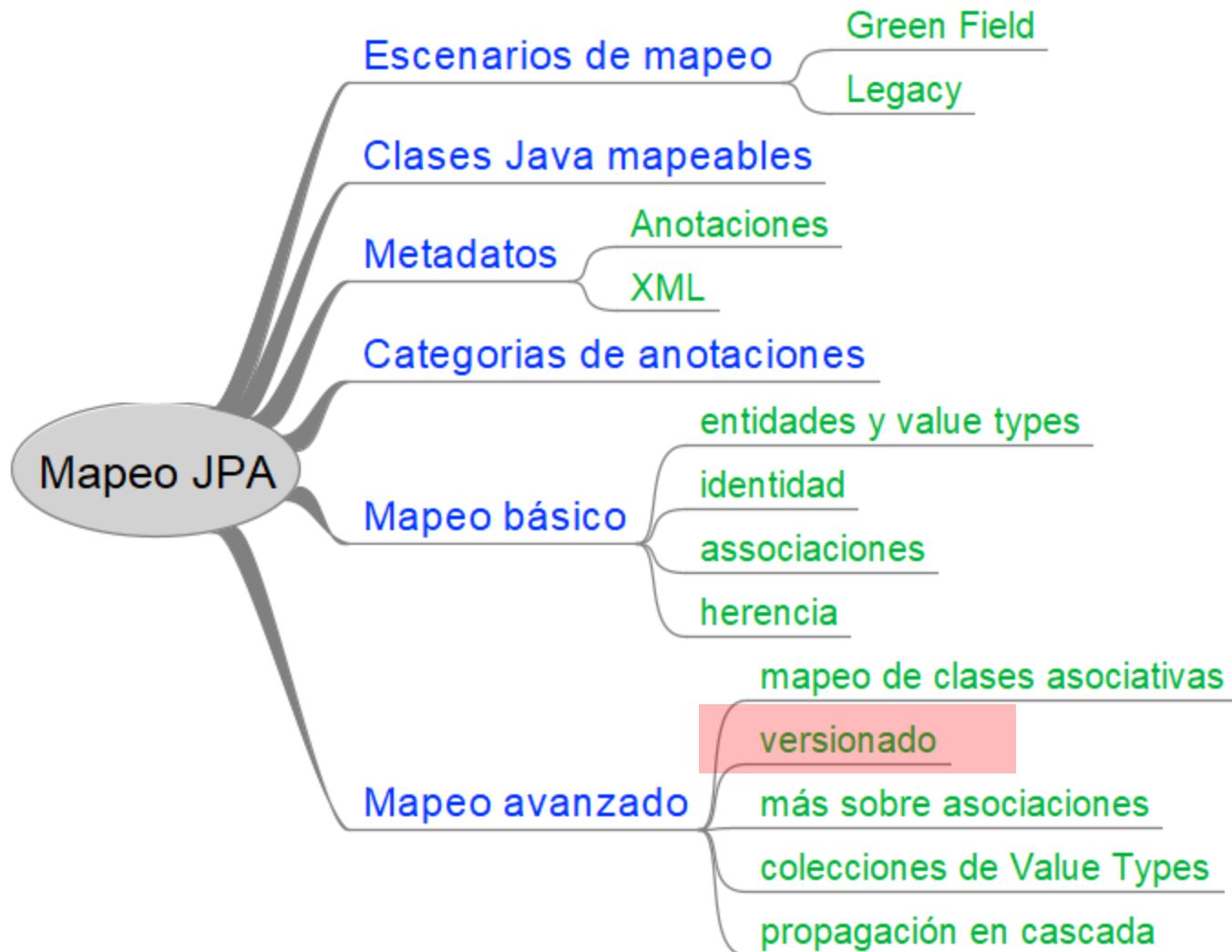
    Charge() {}
}
```

```
@Entity
...
public class Invoice {
    private @Id String id;
    ...
}
```

```
public class ChargeKey implements Serializable {
    private String invoice;
    private String paymentMean;

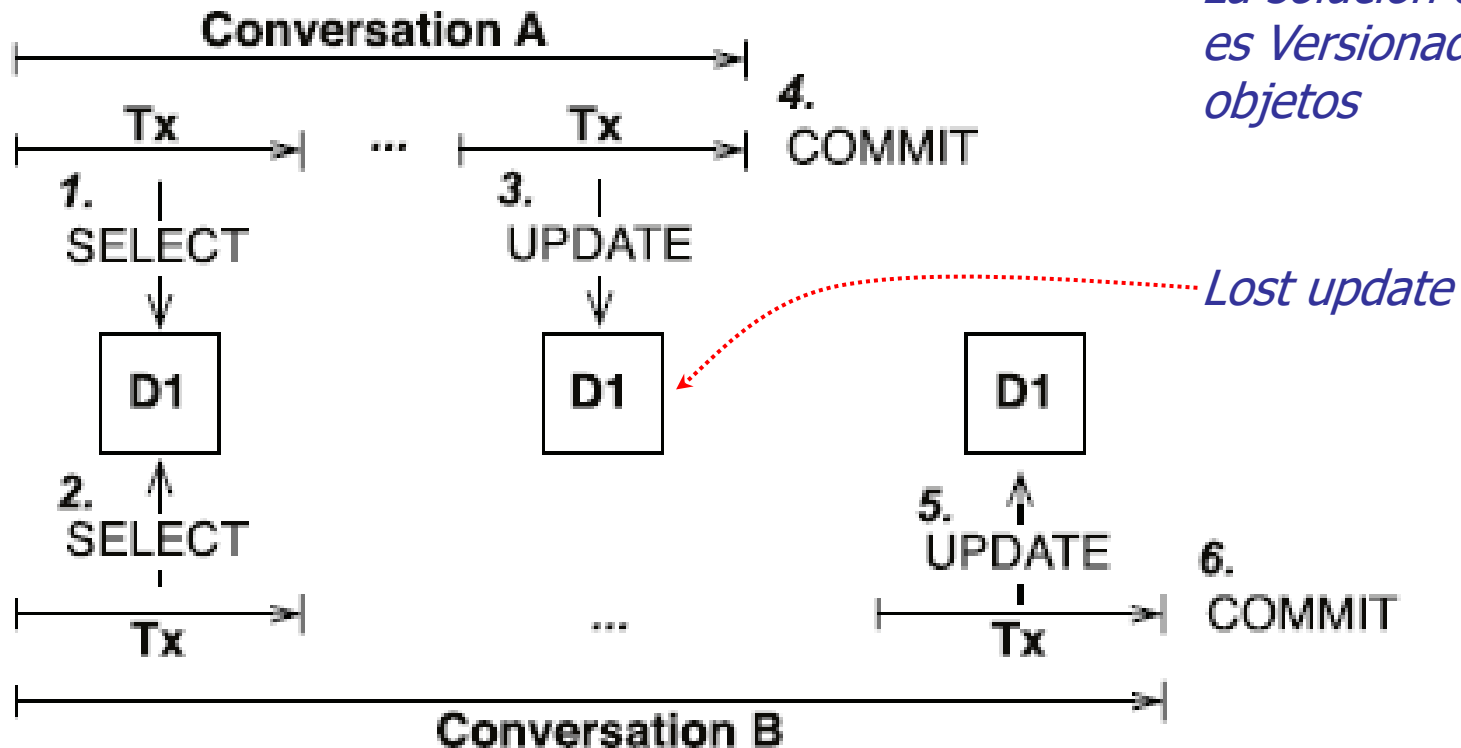
    // hashCode & equals
}
```

```
@Entity
...
public abstract class PaymentMean {
    private @Id String id;
    ...
}
```

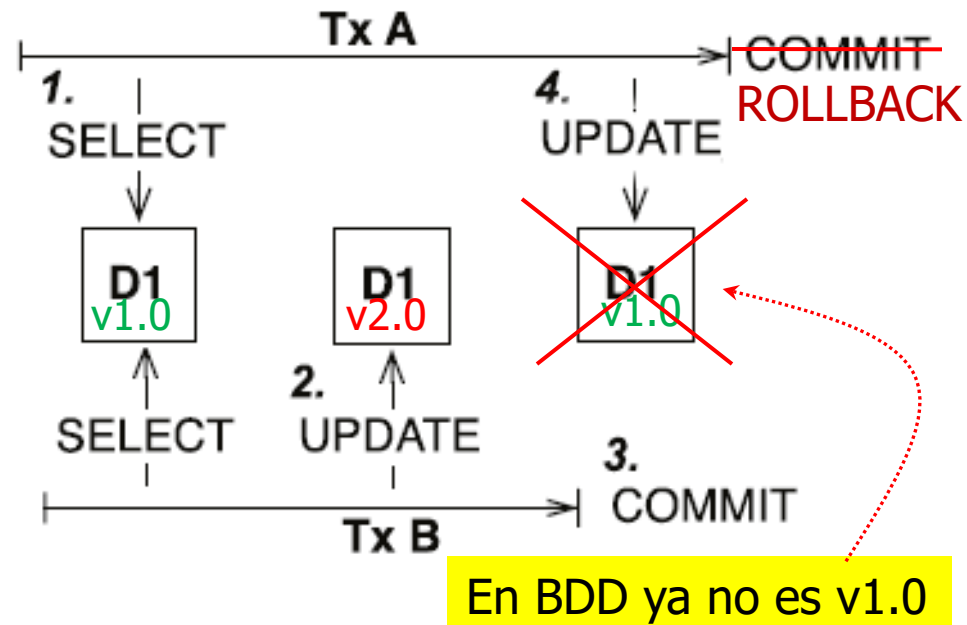
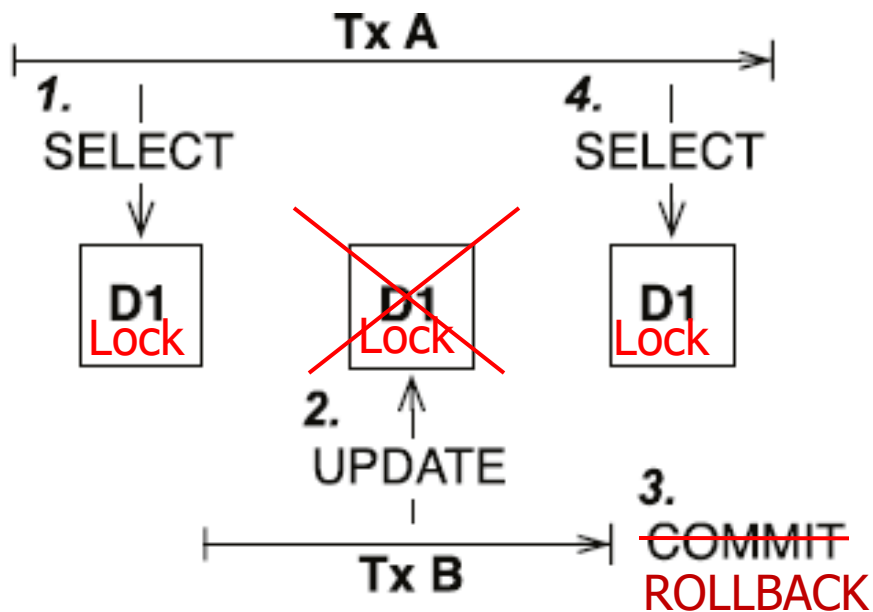


Control de concurrencia optimista

Diferencia entre trx de sistema y trx de aplicación



Control pesimista y optimista



Campo extra en entidades para versión

Añadir información al objeto para poder detectar cambios entre el estado detached y persistent

- Long (contador)
- Timestamp (de la última modificación)

```
@MappedSuperclass
public abstract class BaseEntity {

    @Id private String id = UUID.randomUUID().toString();
    @Version private Long version;
```

El campo versión es actualizado automáticamente por el mapeador cada vez que se actualiza la fila de la tabla

Campo extra versión

El campo versión es mantenido por el mapeador:

- Es actualizado en cada modificación de la fila
- Antes de actualizar comprueba la versión de fila con la versión de la entidad, si no coinciden lanza `OptimisticLockException` (no chequeada)

`java.lang.RuntimeException`

`javax.persistence.PersistenceException`

`javax.persistence.EntityExistsException`

`javax.persistence.EntityNotFoundException`

`javax.persistence.LockTimeoutException`

`javax.persistence.NonUniqueResultException`

`javax.persistence.NoResultException`

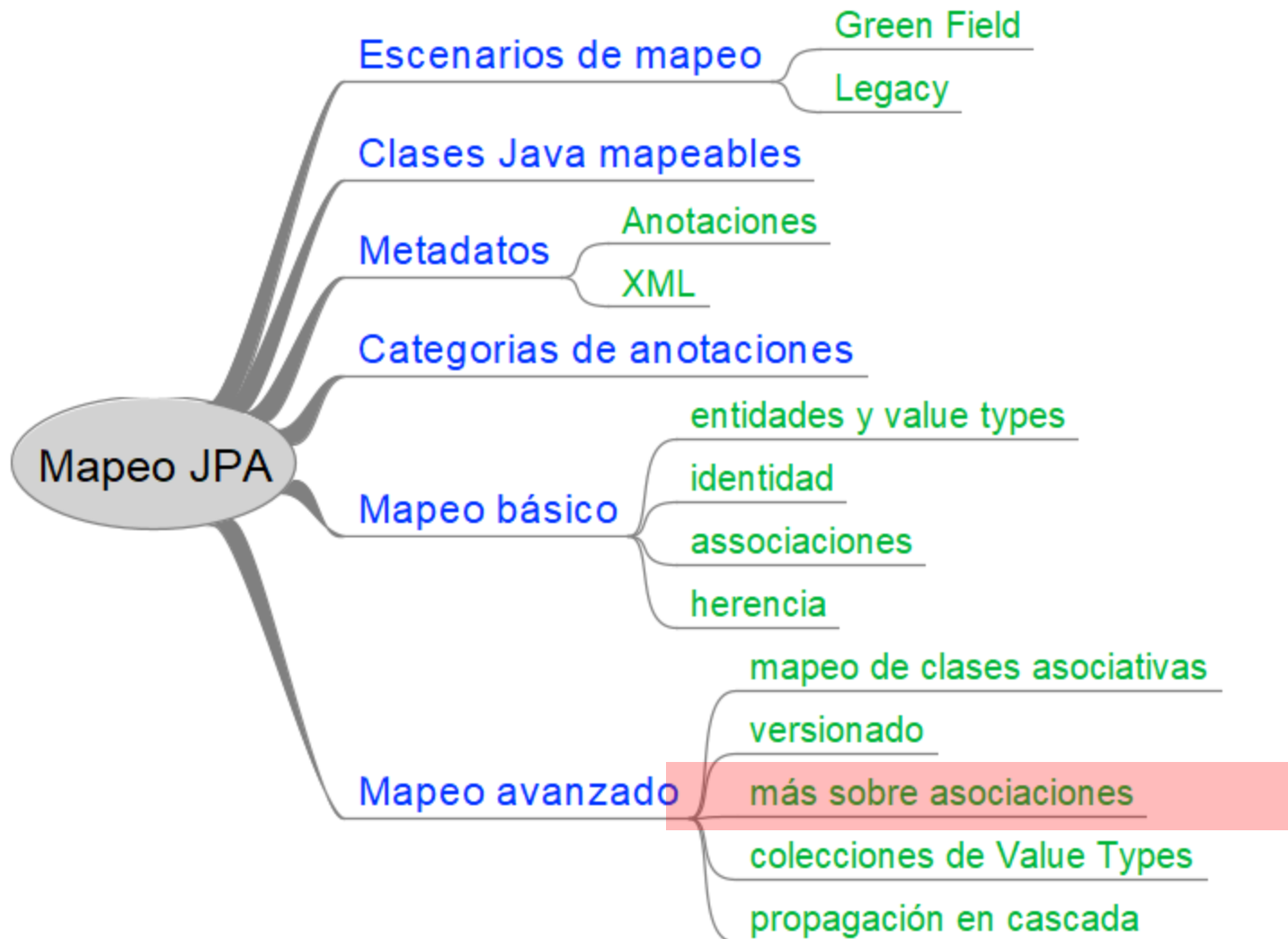
`javax.persistence.OptimisticLockException`

`javax.persistence.PessimisticLockException`

`javax.persistence.QueryTimeoutException`

`javax.persistence.RollbackException`

`javax.persistence.TransactionRequiredException`



Uno a mucho 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;  
    ...  
}
```

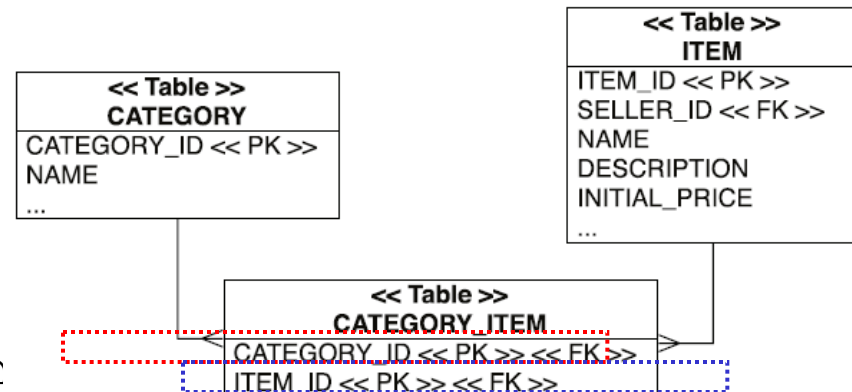
Muchos a muchos

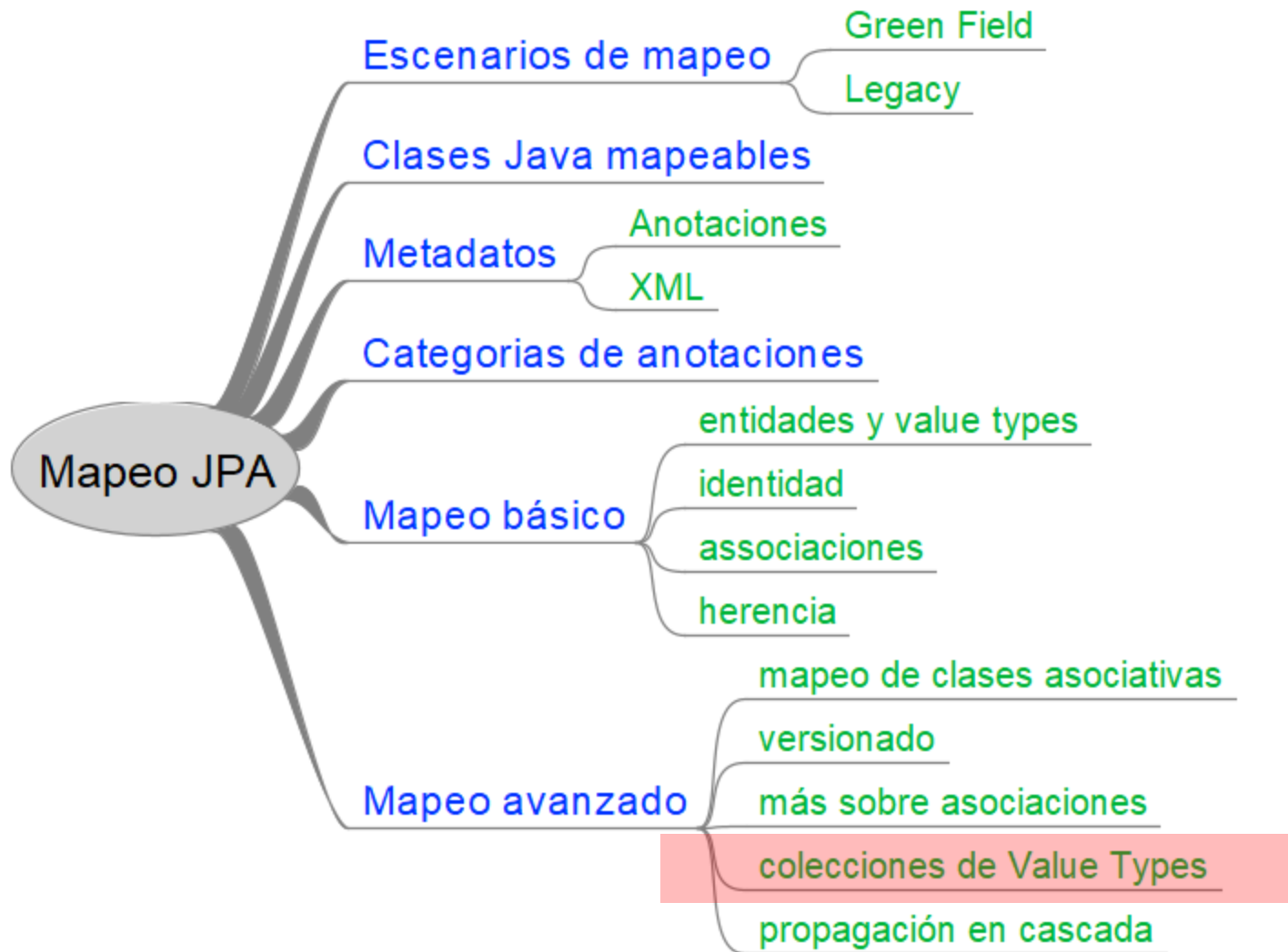
bidireccional

```
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





@ElementCollection

Colecciones de Value Types

- Sets, bags, lists, y mapas de value types
- Colecciones de @Embeddable
- Similar to a OneToMany, pero sobre un ValueType, no sobre entidades

Colecciones de Value Types, limitaciones

No se pueden hacer consultas, guardar o actualizar independientemente de su clase padre

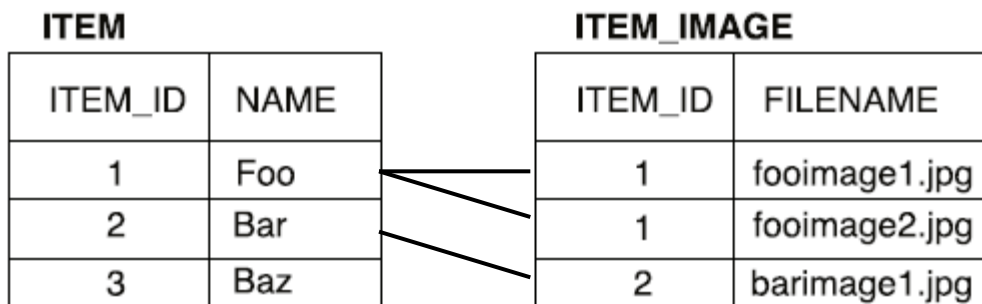
- Strictly privately-owned (dependent) objects.
- Semántica de composición, posesión fuerte

No hay opción de cascada

- Los valuetypes son siempre persistidos, actualizados y eliminados con su clase padre

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

oct.-20

ITEM	
ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE		
ITEM_ID	POSITION	FILENAME
1	0	fooimage1.jpg
1	1	fooimage2.jpg
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

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

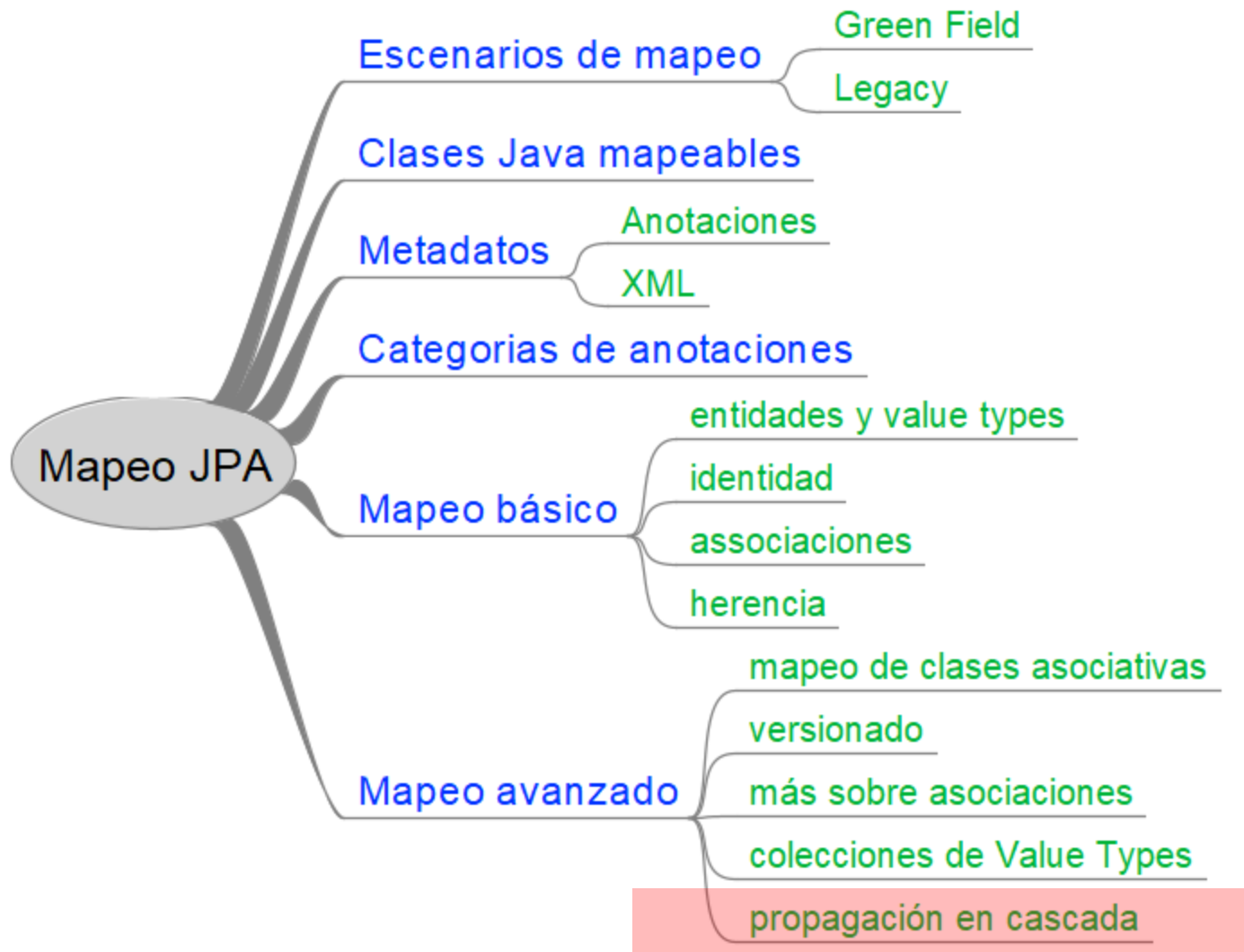
```
@Entity
public class Item {
    @Id @GeneratedValue
    private Long id;
    ...
    @ElementCollection
    @Sort(type=SortType.NATURAL)
    private SortedSet<String> names = new TreeSet<String>();
    ...
}
```

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**



Cascada o persistencia transitiva

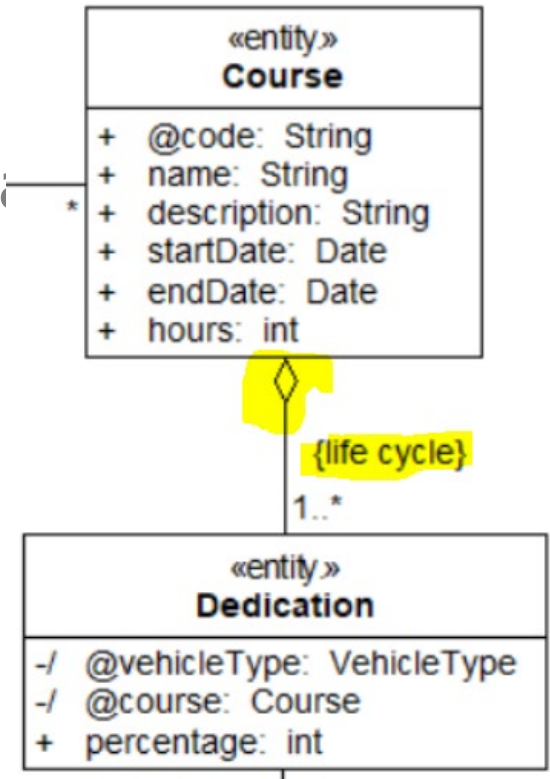
Se da en las relaciones padre/hijo (agregados)

- Los hijos dependen del padre
- Es un caso especial de relación UNO a MUCHOS

*¡Úsalo con mucho cuidado!
O no lo uses, se consigue el mismo efecto programando...*

```
@Entity
public class Course extends BaseEntity {
    @Column(unique=true) private String code;
    ...

    @OneToMany(mappedBy = "course", cascade = CascadeType.ALL)
    private Set<Dedication> dedications = new HashSet<>();
}
```



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>();  
    ...  
}
```