



Mapeo de clases

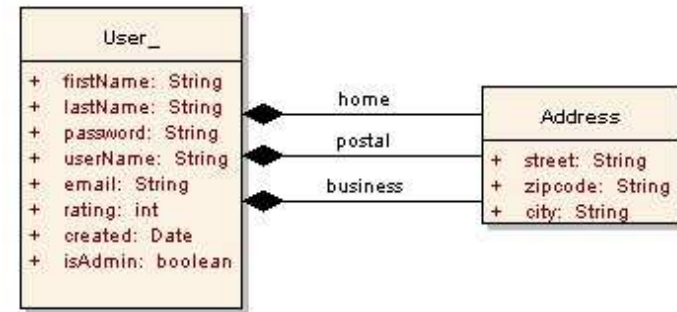
Sistemas de persistencia de
objetos



Entidades

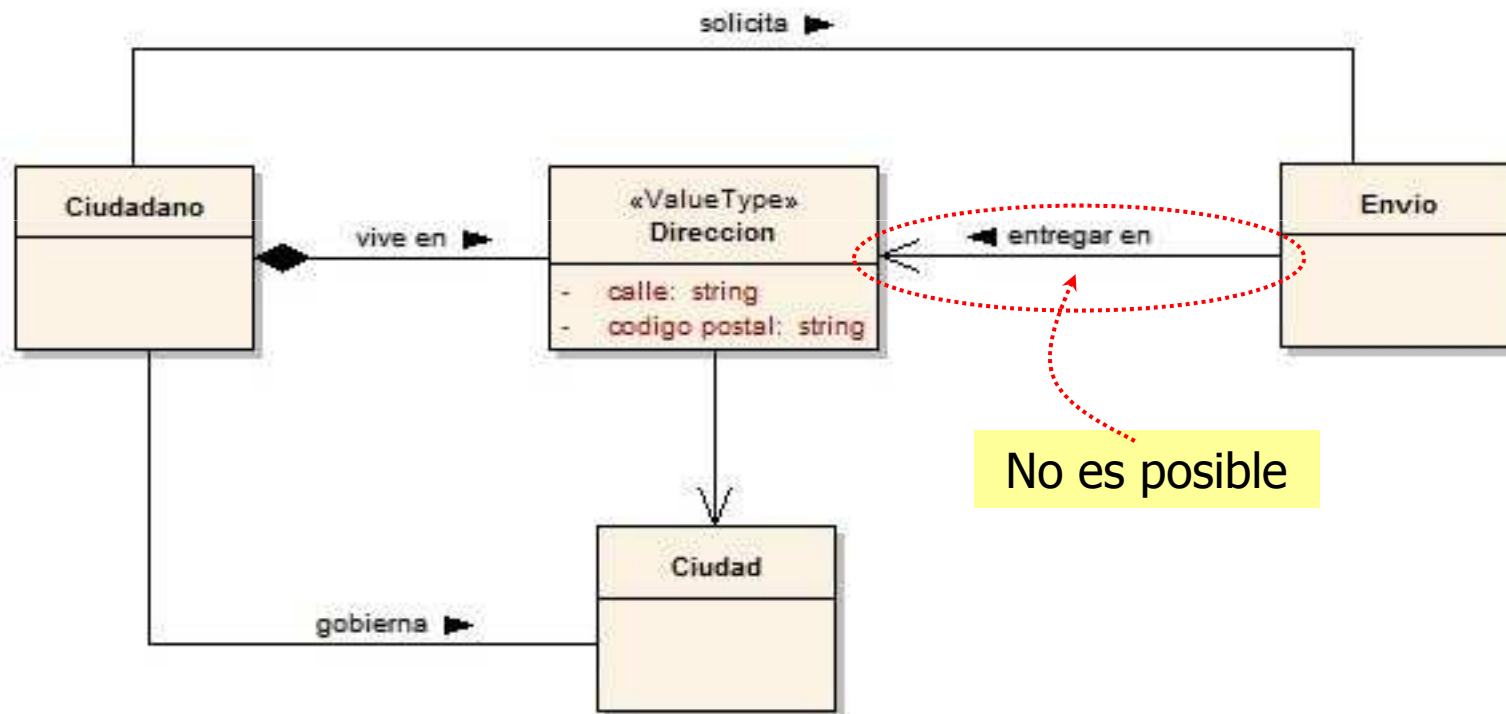
- Un entidad representa un concepto del dominio
- Puede estar asociada con otras entidades
- Su ciclo de vida es independiente
- Debe tener una clave primaria

Value Types

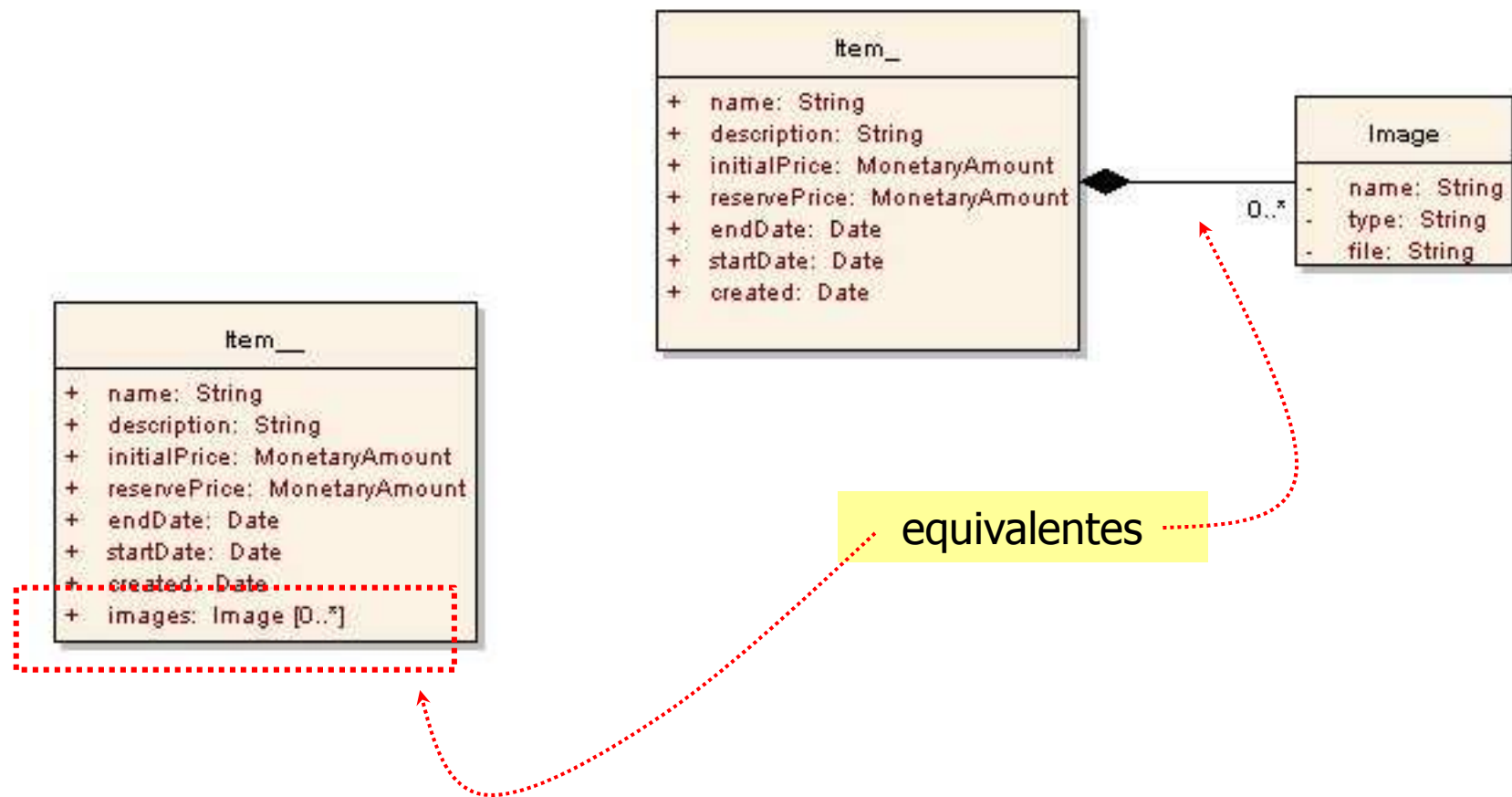


- Representan información adicional, no conceptos principales de dominio
- Se suelen presentar como atributos de una entidad o como composiciones (UML)
- Su ciclo de vida depende enteramente de la entidad que las posee
- No pueden tener referencias entrantes

VT, no referencias entrantes



Representación UML de VT





Representación en Java

```
public class Item {  
    private String name;  
    private String description;  
    private MonetaryAmount initialPrice;  
    private MonetaryAmount reservePrice;  
    private Date endDate;  
    private Date startDate;  
    private Date created;  
  
    private Set<Image> images = new HashSet<Image>();  
  
    . . .  
}
```

Item__
+ name: String
+ description: String
+ initialPrice: MonetaryAmount
+ reservePrice: MonetaryAmount
+ endDate: Date
+ startDate: Date
+ created: Date
+ images: Image [0..*]



Paso de Value Types

- Siempre por copia
 - En java por defecto se pasan referencias
 - Problemas al recibir en setters
 - Cuidado con los getters
- Alternativa: inmutables
 - Las clases básicas del JDK
 - String, Integer, Long, Double, etc

Cuidado en getters

```
public class Item {  
    private String name;  
    . . .  
    private Date endDate;  
    . . .  
    private Set<Image> images = new HashSet<Image>();  
  
    public Date getEndDate() {  
        return endDate;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public Set<Image> getImages() {  
        return images;  
    }  
}
```

Peligro!!!

Seguro, String es immutable

Peligro!!!

Getters pueden romper encapsulación

```
Item item = ItemDAO.getById(1234);  
assert(item.getDate().getHours() == 15);
```

```
Date d = item.getEndDate();  
d.setHours(22); // <-- Peligro
```

```
assert(item.getDate().getHours() == 15); // false  
assert(item.getDate().getHours() == 22); // true
```

```
public Date getEndDate() {  
    return endDate.clone();  
}
```

```
Item item = ItemDAO.getById(1234);  
assert(item.getImages().size() == 15);
```

```
Item.getImages().clear();
```

```
assert(item.getImages().size() == 15); // false  
assert(item.getImages().size() == 0); // true
```

```
public Set<Image> getImages() {  
    return Collections.unmodifiableSet(images);  
}
```



POJO (plain old java objects)

- 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
 - Hibernate nativo → xml, hibernate annotations
 - JPA → 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



POJOs 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
 - 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

- Todos tipos JDK tienen persistencia automática
- Campos de otro tipo:
 - Referencias a ValueTypes: si son de clases @Embeddable todos los campos a la misma tabla
 - Referencias a Entidades: son relaciones, no campos. FK a la tabla de @Entity
 - Resto de casos, serialización
 - Debe implementar [Serializable](#)

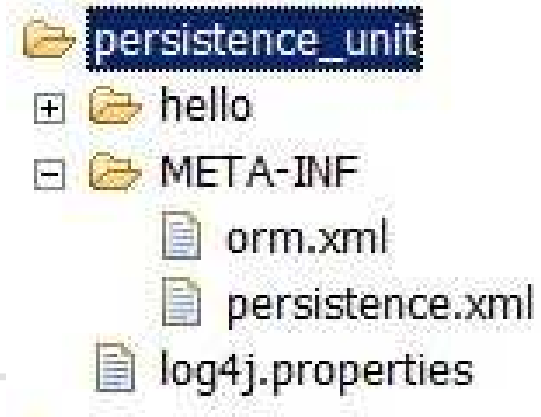


Metadatos en annotations

- @Entity → entidades
- @Embeddable → Value Types
- La posición de @Id determina el modo de acceso del motor de persistencia a los atributos
 - Acceso "field" (`public`, `private`, `protected`, `package`)
 - Acceso "properties" (a través de get/set)
 - getters y setters `public` o `protected`



Metadatos en XML



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



Metadatos xml, ejemplo

```
<entity-mappings>
  <entity class="auction.model.Item" access="FIELD">
    ...
    <one-to-many name="bids" mapped-by="item">
      <cascade>
        <cascade-persist/>
        <cascade-merge/>
        <cascade-remove/>
      </cascade>
    </one-to-many>
  </entity>
  <entity class="auction.model.Bid" access="FIELD">
    ...
    <many-to-one name="item">
      <join-column name="ITEM_ID"/>
    </many-to-one>
  </entity>
</entity-mappings>
```



Categorías de anotaciones

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



Anotaciones por categoría

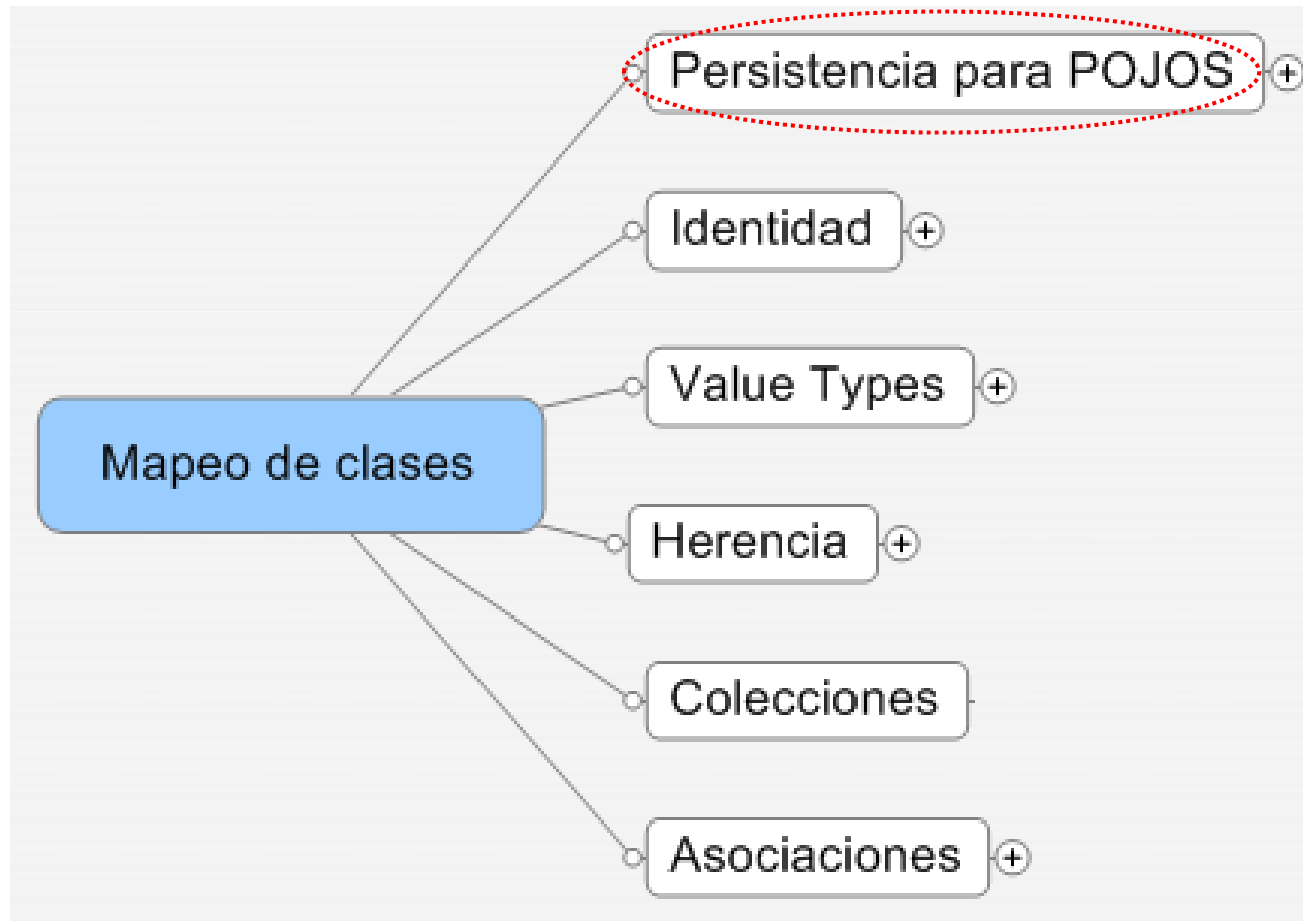
Category	Annotations
Entity	<u>@Entity</u>
Database Schema Attributes	<u>@Table</u> <u>@SecondaryTable</u> <u>@SecondaryTables</u> <u>@Column</u> <u>@JoinColumn</u> <u>@JoinColumns</u> <u>@PrimaryKeyJoinColumn</u> <u>@PrimaryKeyJoinColumns</u> <u>@JoinTable</u> <u>@UniqueConstraint</u>
Identity	<u>@Id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

Anotaciones por categoría

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

Mapeo de clases





Entidades

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

- @Entity
 - Marca una clase como entidad
 - Atributo "name" opcional → será el usado en las queries
- @Table

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;
    ...
}
```


Attribute	Required	Description
name		De la comuna en la tabla
unique		Default: false. Estable un índice único en la columna
nullable		Default: true. ¿El campo admite nulos?
insertable		Default: true. Estable si la columna aparecerá en sentencias INSERT generadas
updatable		Default: true. ¿Incluido en SQL UPDATE?
columnDefinition		Default: empty String. Fragmento SQL que se empleará en el DDL para definir esta columna.
table		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 @SecondaryTable
length		Default: 255 para String. Longitud de los campos string.
precision		Default: 0 (sin decimales). Cantidad de decimales.
scale		Default: 0.



@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



- 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

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



@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

- @Temporal
 - Matiza el formato final de los campos `java.util.Date` y `java.util.Calendar`
 - En la BDD serán DATE, TIME o TIMESTAMP
 - Opciones: DATE, TIME, TIMESTAMP

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



@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;
    ...}

```

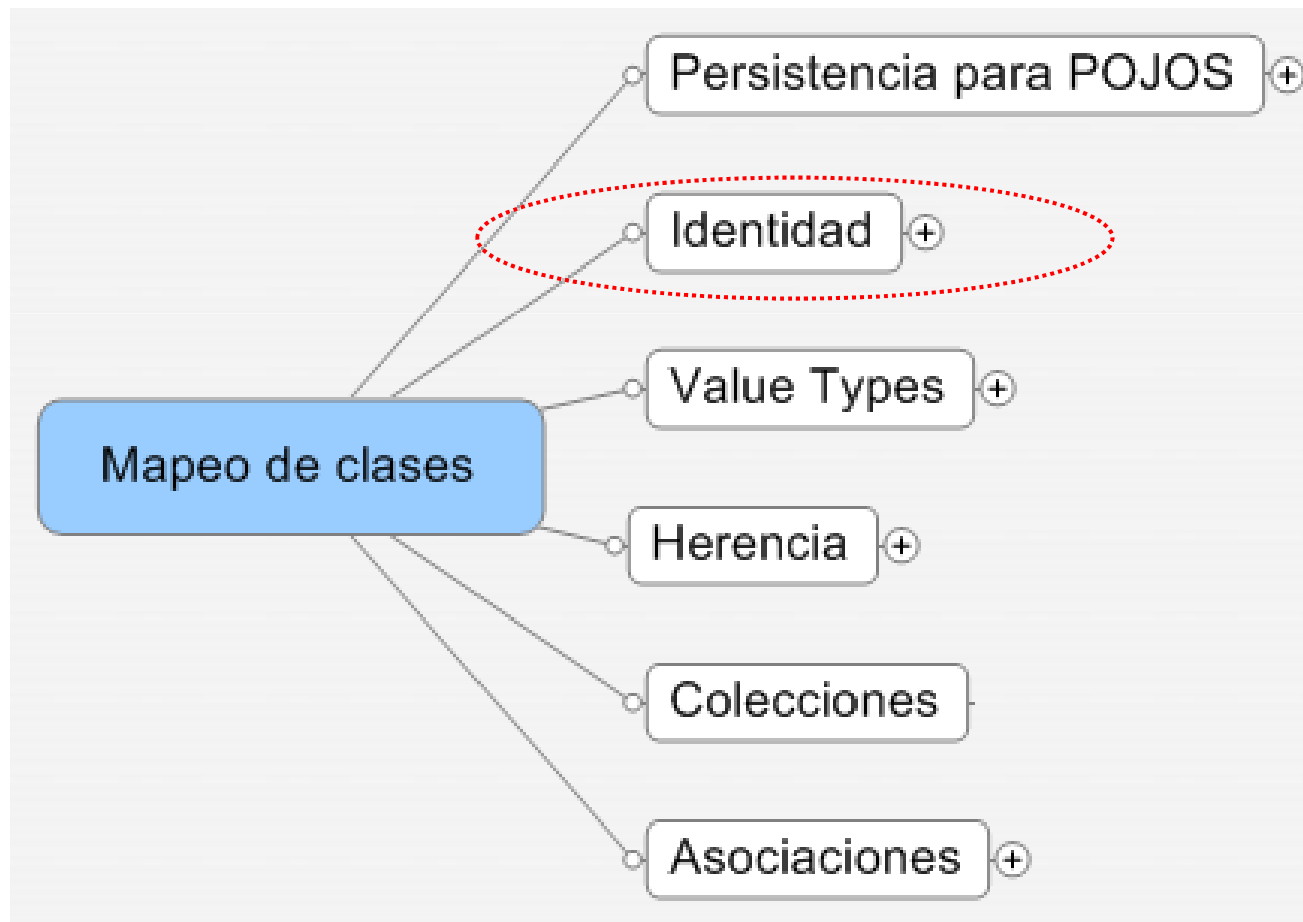
Tabla de tipos hibernate

De tipo Java	A tipo SQL
integer, long, short, float, double, character, byte, boolean, yes_no, true_false	
Tipos básicos java	Tipos SQL básicos de cada vendedor
string	
java.lang.String	VARCHAR (o Oracle VARCHAR2)
date, time, timestamp	
java.util.Date y sus subclases	DATE, TIME y TIMESTAMP
calendar, calendar_date	
java.util.Calendar	DATE, TIME y TIMESTAMP
big_decimal, big_integer	
java.math.BigDecimal y java.math.BigInteger	NUMERIC (Oracle NUMBER)

Tabla de tipos hibernate (2)

big_decimal, big_integer		
java.math.BigDecimal y java.math.BigInteger		NUMERIC (Oracle NUMBER)
locale, timezone, currency		
java.util.Locale java.util.TimeZone java.util.Currency		VARCHAR (o Oracle VARCHAR2)
class		
java.lang.Class		VARCHAR (o Oracle VARCHAR2) sólo el nombre de la clase
binary		
Byte arrays		Tipo binario de la BBDD
text		
java.lang.String muy largas		CLOB o TEXT

Mapeo de clases





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 identificador, usualmente un `Long`
- No siempre serán iguales
 - El periodo de tiempo que sí lo son se le denomina "Ámbito de identidad garantizada", ó "Ámbito de persistencia"

Identidad de BBDD

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

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

@Entity

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

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**



Tipos de claves

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

¿Cual es mejor para
formar la clave primaria?




Clave candidata

- Condiciones
 - 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
 - ¿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 ...



Business keys

- Podrían ser claves candidatas
- Pero existe la probabilidad (baja) de que su valor cambie en el tiempo
- Son de utilidad para el usuario ya que las emplea de forma cotidiana
 - P.e. el nombre del departamento (puede cambiar pero pocas veces)
- No sirven como clave primaria pero tienen su utilidad

Claves artificiales (surrogate keys)

- Sin significado en el contexto
- Siempre generadas por el sistema
- Varias estrategias de generación

JPA

- AUTO → JPA selecciona según BBDD
- IDENTITY
- SEQUENCE
- TABLE

Ver documentación de referencia

Hib

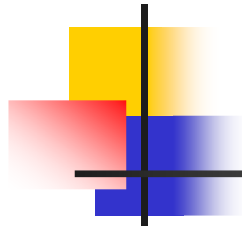
- hi/lo
- uuid.hex
- Nuevas implementando un interfaz

genera string de 32 caracteres
único en el mundo

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

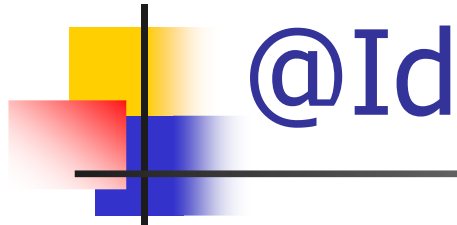
110V-00

alb@uniovi.es



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
- Las claves candidatas se hacen UNIQUE
- Las candidatas (y si no hay, las business keys) son las que se emplean en el **equals()**



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

- En cada entidad al menos:
 - Una @Id
 - Multiple @Id y una @IdClass para la clase que forma clave (clave compuesta)
 - Una @EmbeddedId

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

alb@ur

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

    public Employee() { }

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

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

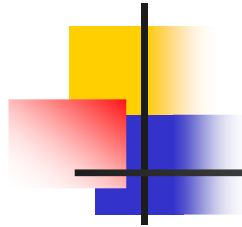


@GeneratedValue

- 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		String, el nombre relaciona el generador caracterizado con @SequenceGenerator o @TableGenerator

El problema del equals() y hashCode()



```
Map<User, Item> users = new HashMap<User, Item>();
```

```
User u = new User();  
u.setFirstname("Pepe");
```

```
Item i = new Item();
```

```
users.put(u, i);  
u.setFirstname("Otro nombre");  
Item ii = users.get(u);
```

Suponiendo que el `hashCode()` y el `equals()` se calculan incluyendo el nombre del usuario

¿ `i == ii` ?



HashCode() y equals()

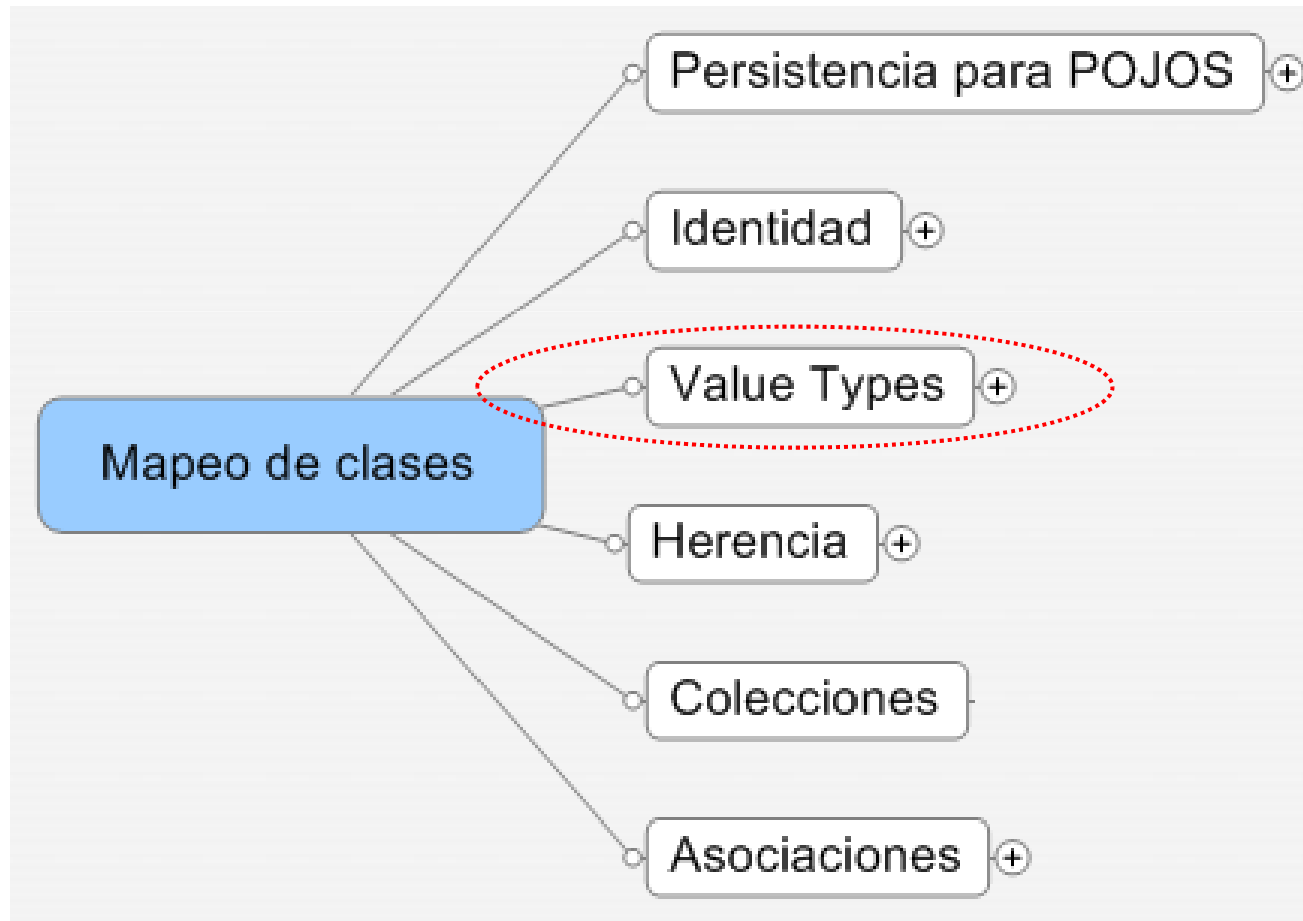
- Los `Set()` y `Map()` emplean `hashCode()` y `equals()` para insertar y luego localizar los elementos en tablas hash.
- Si cambian atributos después de haber introducido un objeto en `Set()` o `Map()` no se volverá a encontrar o se podrán insertar repetidos
- El contrato de uso de `Set()` y `Map()` **exige** que no se cambien los atributos del objeto mientras están en la colección
- ¿Sobre qué atributos hay que definir `hashCode()` y `equals()`?



HashCode() y equals()

- No se pueden definir sobre las claves artificiales ya que **no existen** hasta que no se inserta en la BBDD
 - Se generan allí, al hacer el INSERT
 - Al final de la transacción
- Se deben definir sobre los atributos que forman una clave candidata o en su defecto sobre una business key

Mapeo de clases





Entities vs Value types

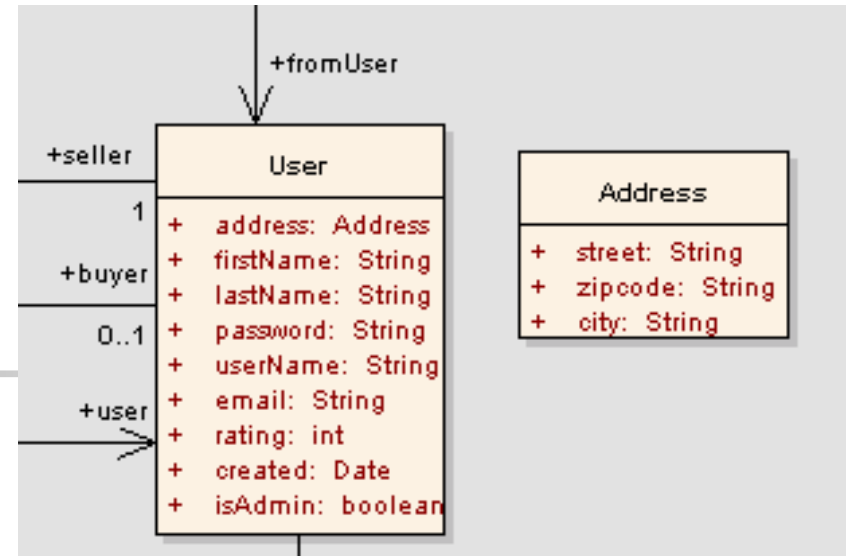
- Una de las riquezas de los modelos OO
→ más clases que tablas
- **Entidades** son aquellas clases de las cuales los objetos son relevantes para el usuario
 - En JPA siempre llevan identificador y deben ajustarse a un convenio de nombres (mínimo)



Entities vs Value types

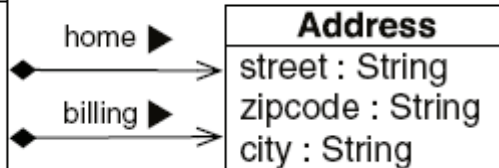
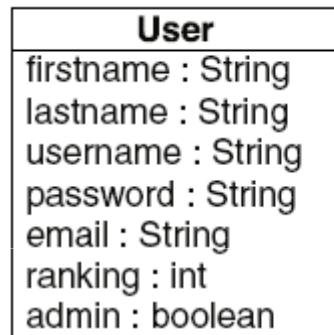
- **Tipos valor** son aquellas clases cuya identidad no es conocida por el usuario, ni le importa
 - Tienen semántica de composición o directamente aparecen como atributos en los diagramas UML
 - Las clases JDK (Integer, Long, etc.) son de este tipo
 - Su ciclo de vida depende totalmente de la entidad a la que están asignados
 - Los objetos Valor sólo tienen referencias entrantes de los objetos que los poseen y no pueden ser compartidos con otros objetos
- La diferencia entre uno y otro es difícil de definir ya que depende del contexto

Referencias



- A entidades
 - Se salvan como claves ajenas
- A value types
 - Se salvan en la misma tabla excepto si son colecciones (p.e. un usuario tiene varias direcciones)
 - Se usa la etiqueta **@Embedded**

Ejemplo



```

public class Address {

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

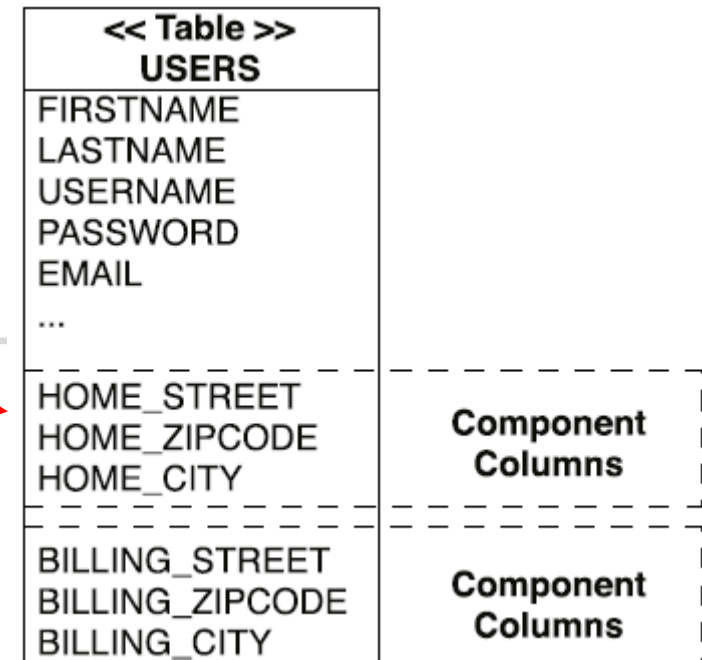
    public Address() {}

    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; }

```





Mapeos

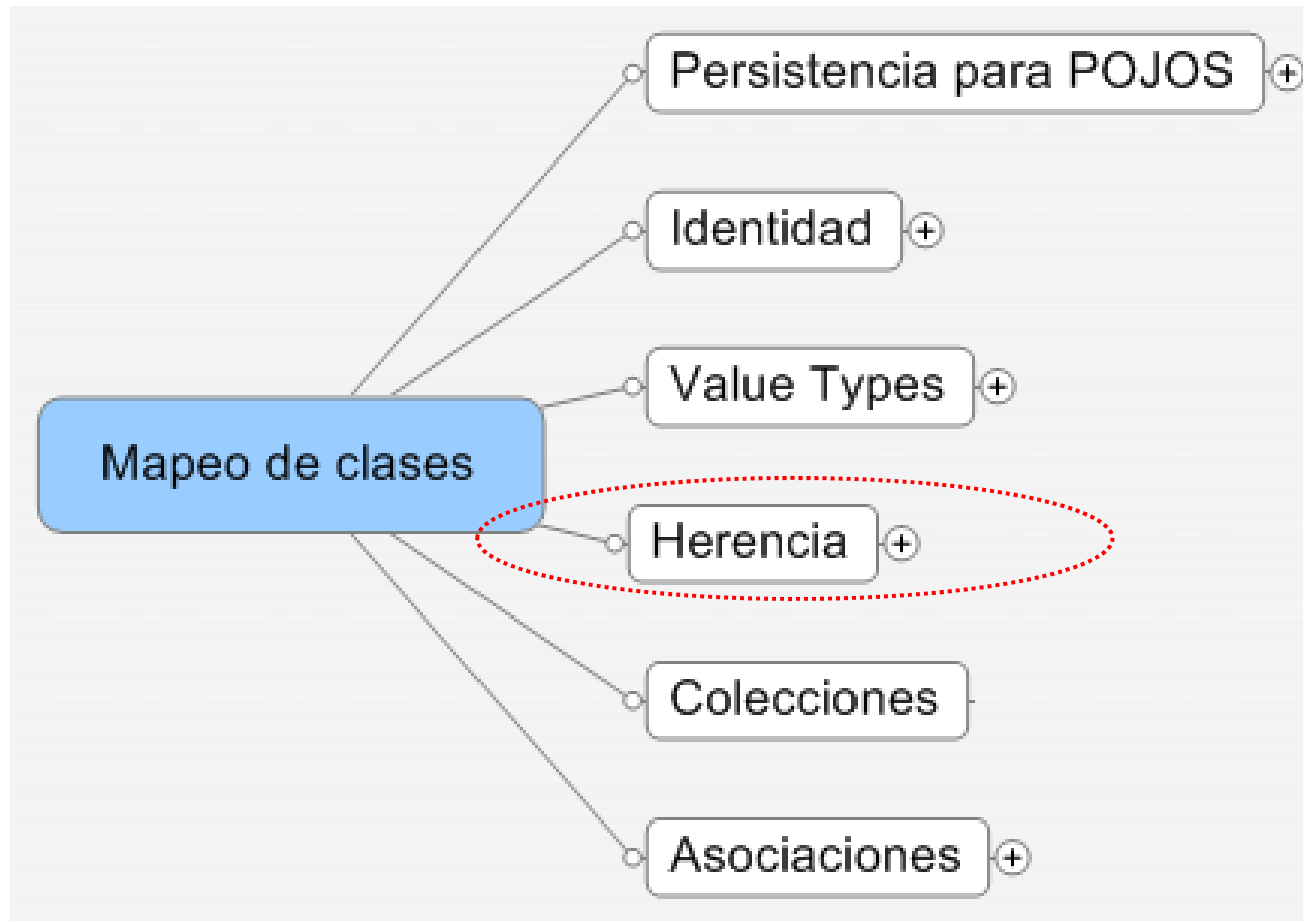
```
@Entity
@Table(name = "USERS")
public class User {
    ...
    @Embedded
    private Address homeAddress;

    @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;
    ...
}
```

Mapeo de clases





Estrategias para mapear herencia

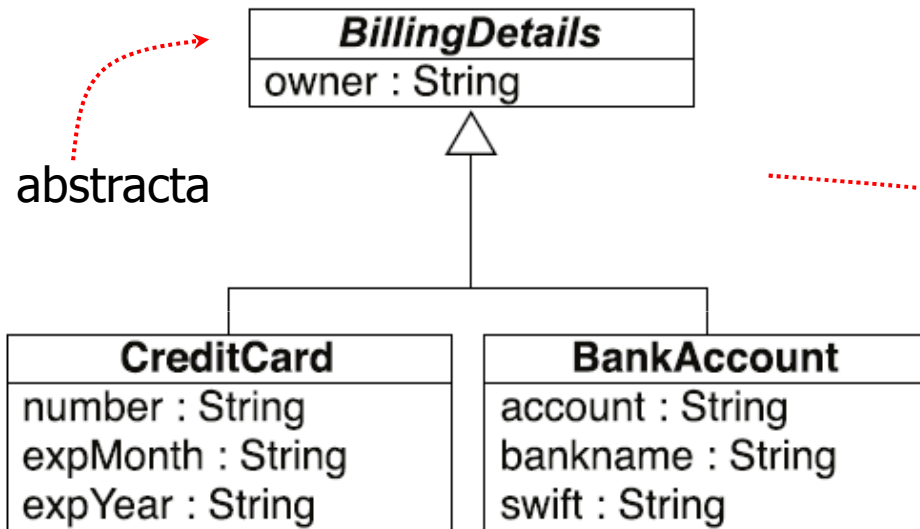
- JPA permite 3
- Tabla por cada clase no abstracta
 - `InheritanceType.TABLE_PER_CLASS`
- Tabla por cada clase
 - `InheritanceType.JOINED`
- Tabla única para toda la jerarquía
 - `InheritanceType.SINGLE_TABLE`



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;
    ...
}

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

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



Table per class hierarchy

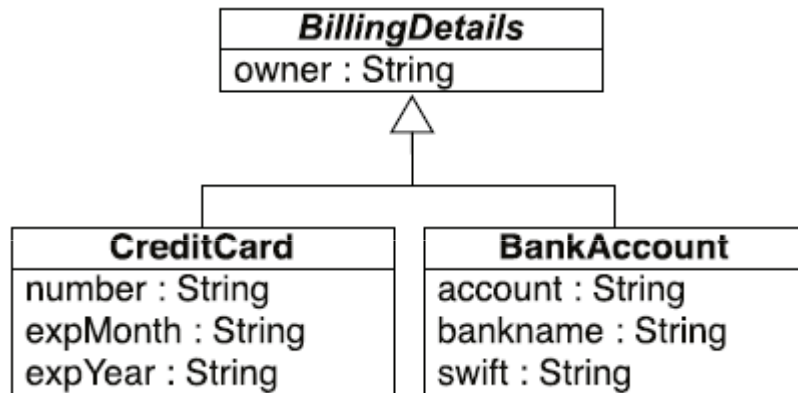
- Todas las clases persisten en una única tabla con la unión de todas las columnas de todas las clases
- Usa un discriminador en cada fila para distinguir el tipo
- Ventajas
 - Es simple y eficiente
 - Soporta el polimorfismo
 - Fácil de implementar
 - Fácil modificar cualquier clase
- Desventaja
 - Todas las columnas no comunes deben ser nullables
 - Pueden quedar columnas vacías



Table per class hierarchy (2)

- Mapeo
 - En la clase raiz 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 (3)



<< Table >> BILLING_DETAILS	
BILLING_DETAILS_ID	
BILLING_DETAILS_TYPE << Discriminator >>	
OWNER	
CC_NUMBER	
CC_EXP_MONTH	
CC_EXP_YEAR	
BA_ACCOUNT	
BA_BANKNAME	
BA_SWIFT	

Table per class hierarchy (4)

```
@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;
    ...
}

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

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



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 mas complicadas
 - Para jerarquías muy complejas el rendimiento en consultas puede ser pobre, muchas joins



Table per subclass (2)

- 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 (3)

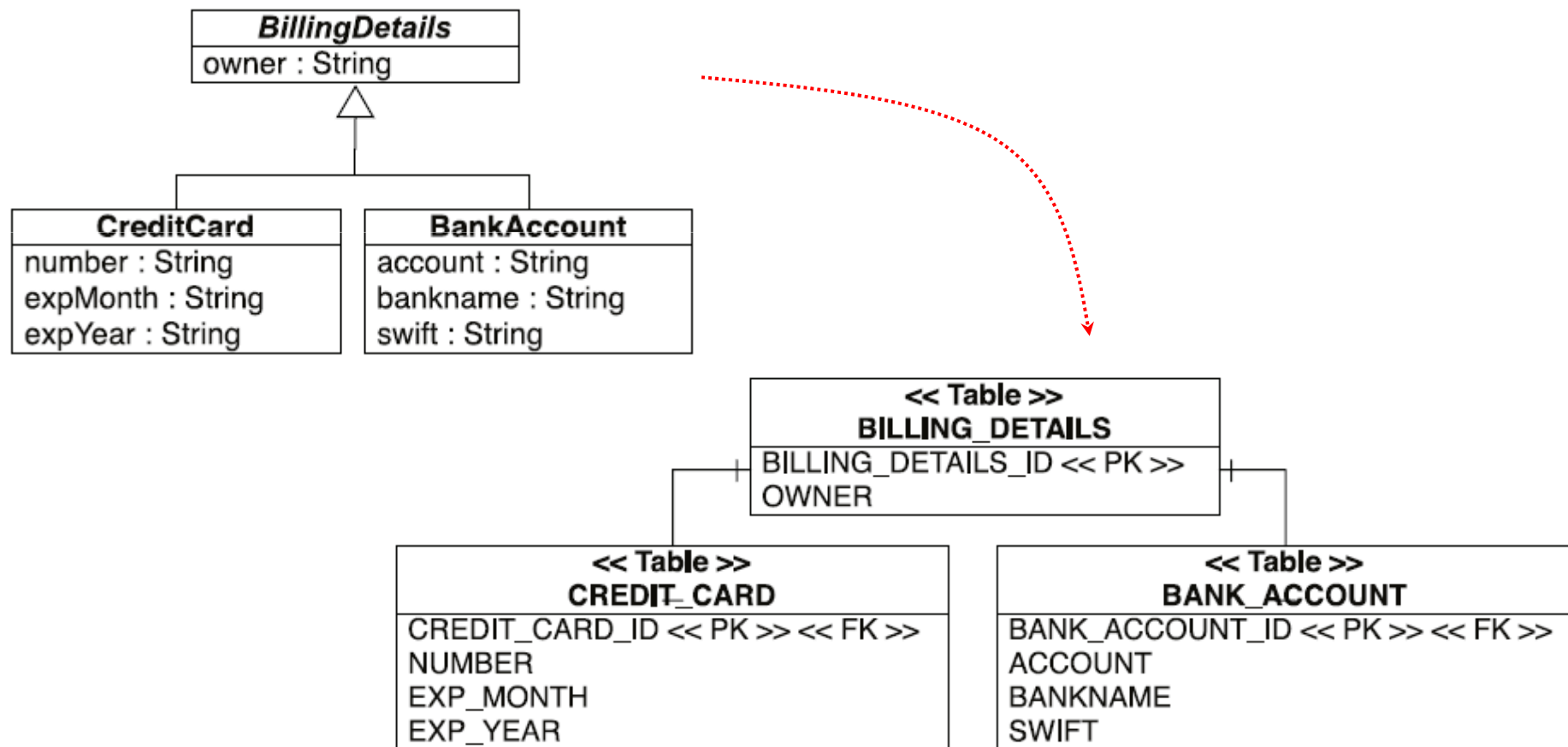


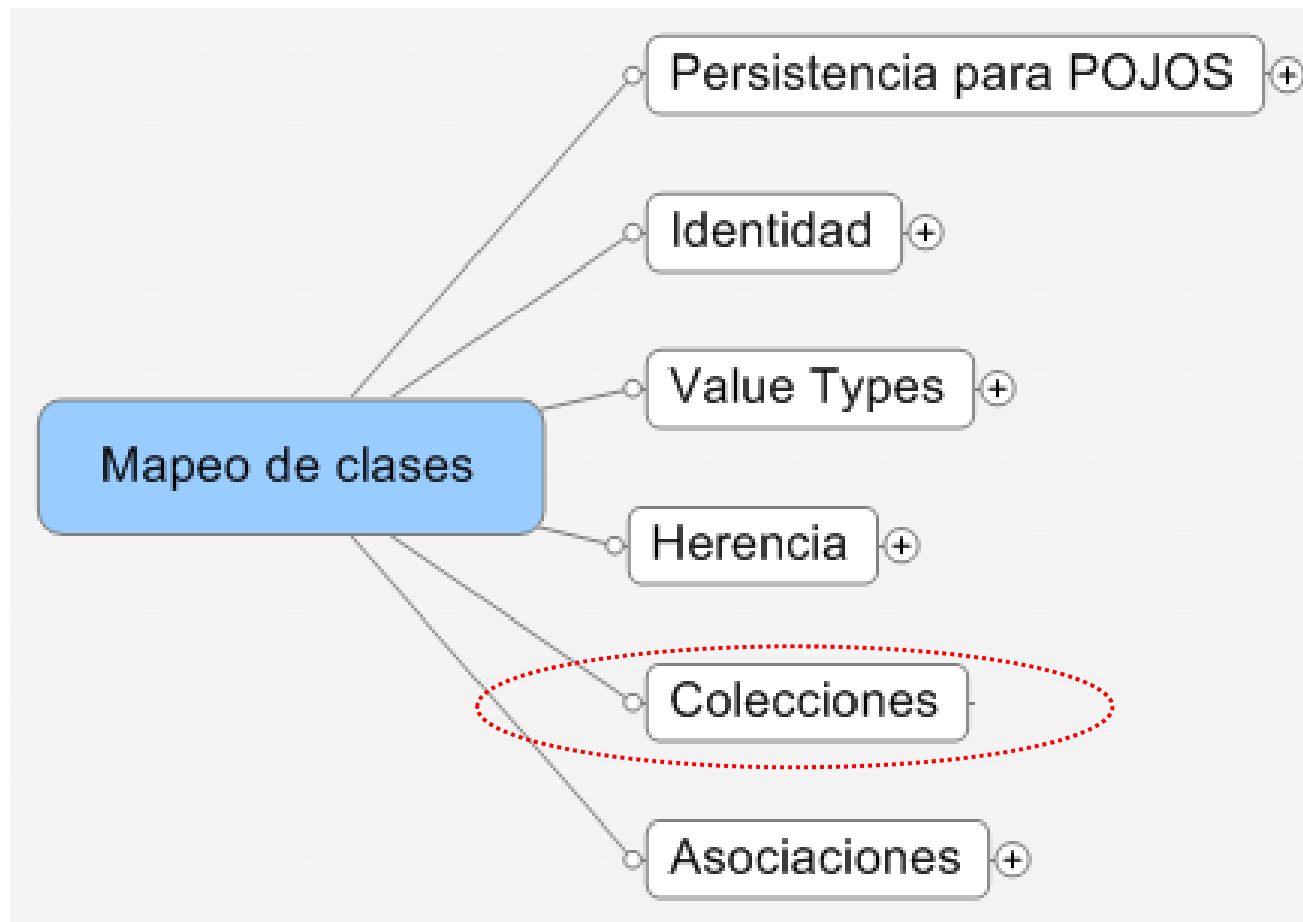


Table per subclass (4)

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

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

Mapeo de clases





Colecciones de Value Types

- No existen en JPA, solo hibernate
- Sets, bags, lists, y maps de value types
- Forma estándar (idiom) en hibernate 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 y mapeos hibernate

Interfaz	Etiqueta de mapeo	Implementación
java.util.Set	<set>	java.util.HashSet
java.util.SortedSet	<set>	java.util.TreeSet
java.util.List	<list>	java.util.ArrayList
java.util.Collection	<bag> o <idbag>	java.util.ArrayList
java.util.Map	<map>	java.util.HashMap
java.util.SortedMap	<map>	java.util.TreeMap
Arrays	<primitive-array>	

	Permite Duplicados	Preserva Orden
java.util.Set	NO	NO
java.util.SortedSet	NO	SI
java.util.List	SI	SI
java.util.Collection	SI	NO
java.util.Map	NO	NO
java.util.SortedMap	NO	SI
Arrays	SI	SI

Lo más usado
para colecciones

Mapeo de Set

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}
public void setImages(Set images) {
    this.images = images;
}
```

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@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)

Mapeo de List

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.IndexColumn(
    name="POSITION", base = 1
)
@Column(name = "FILENAME")
private List<String> images = new ArrayList<String>();
```

@Column, @JoinTable
opcionales

Perserva el
orden

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

Mapeo de Bag

```
private Collection images = new ArrayList();  
...  
public Collection getImages() {  
    return this.images;  
}  
public void setImages(Collection images) {  
    this.images = images;  
}
```

```
@org.hibernate.annotations.CollectionOfElements  
@org.hibernate.annotations.CollectionId(generator="identity",  
    type = @org.hibernate.annotations.Type(type="long"),  
    columns = @Column(name="ITEM_IMAGE_ID")  
)  
@Column(name="IMAGE_FILE_NAME", nullable=false)  
private Collection<String> images = new ArrayList<String>(); // A bag
```

Clave artificial para hacer cada fila
única (bag permite duplicados)

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

nov-

Mapeo de Map

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.MapKey(
    columns = @Column(name = "IMAGENAME")
)
@Column(name = "FILENAME")
private Map<String, String> images = new HashMap<String, String>();
```

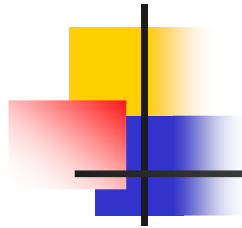
@Column, @JoinTable
opcionales

Guarda las claves
del mapa

ITEM	
ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE		
ITEM_ID	IMAGENAME	FILENAME
1	Image One	fooimage1.jpg
1	Image Two	fooimage2.jpg
1	Image Three	foomage3.jpg

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



Sorted & ordered cols.

- En hibernate no es lo mismo
 - **Sorted** se hace en memoria (JVM) usando interfaz **Comparable**
 - **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

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.Sort(
    type = org.hibernate.annotations.SortType.NATURAL
)
private SortedSet<String> images = new TreeSet<String>();
```

Solo para Set y Map
(se hace en JVM)

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.MapKey(
    columns = @Column(name="IMAGENAME")
)
@Column(name = "FILENAME")
@org.hibernate.annotations.Sort(
    type = org.hibernate.annotations.SortType.NATURAL
)
private Map<String, String> images = new HashMap<String, String>();
```

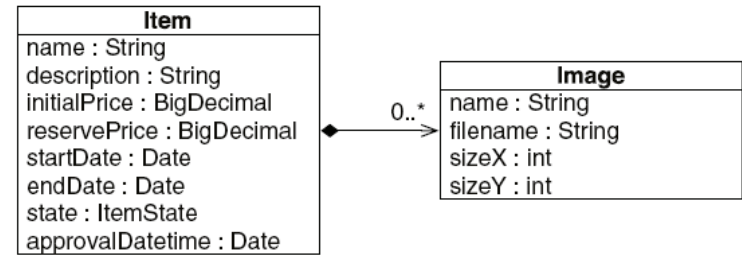


Ordered collections

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.OrderBy(
    clause = "FILENAME asc"
)
private Set<String> images = new HashSet<String>();
```

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

Colecciones de componentes



```

@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@AttributeOverride(
    name = "element.name",
    column = @Column(name = "IMAGENAME", nullable = false)
)
private Set<Image> images = new HashSet<Image>();
    
```

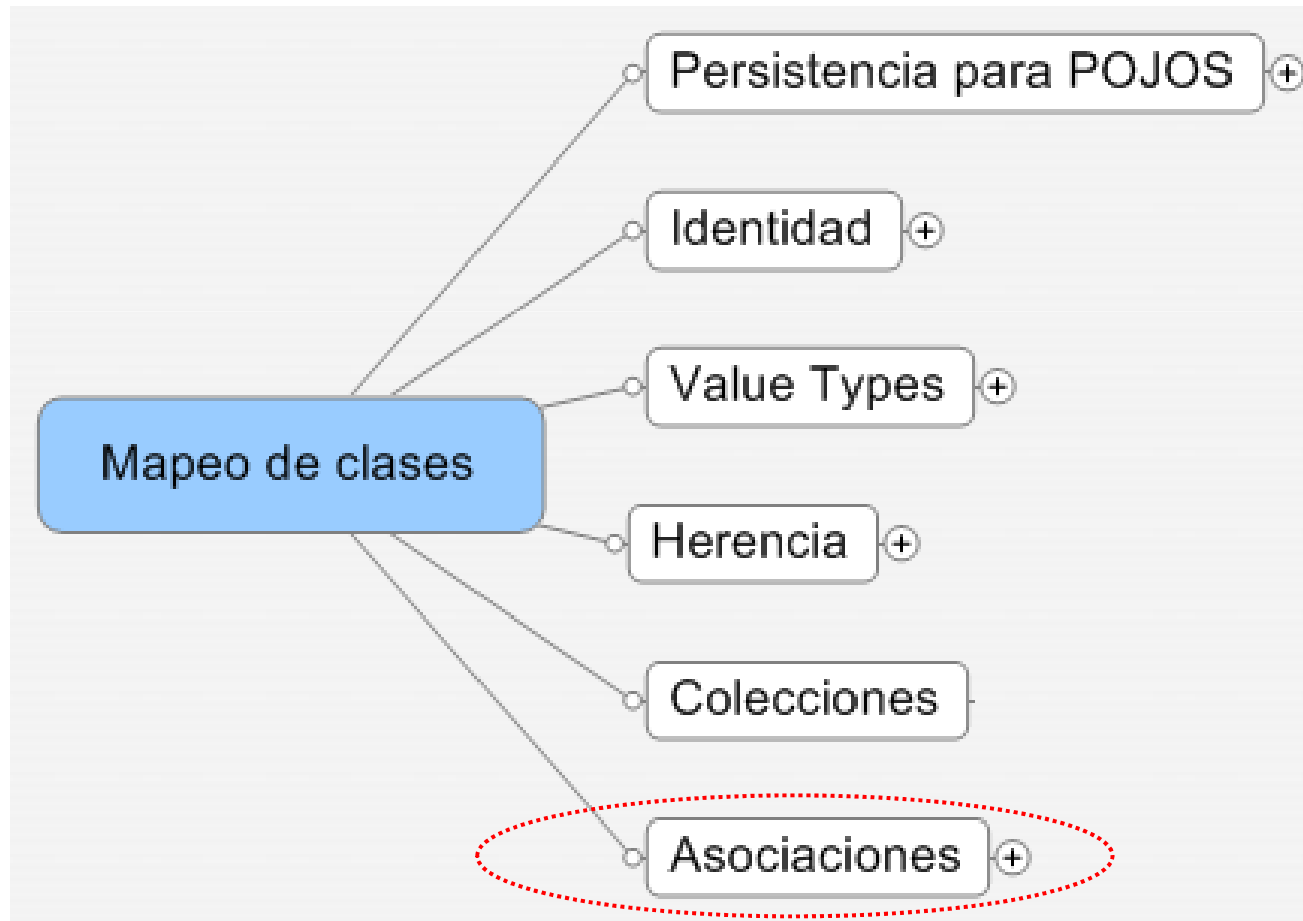
```

@Embeddable
public class Image {
    @Column(nullable = false)
    private String name;
    @Column(nullable = false)
    private String filename;
    @Column(nullable = false)
    private int sizeX;
    @Column(nullable = false)
    private int sizeY;
    ... // Constructor, accessor
}
    
```

Esta clase debe tener redefinidos `hashCode()` y `equals()`

ITEM		ITEM_IMAGE				
ITEM_ID	ITEM_NAME	ITEM_ID	IMAGENAME	FILENAME	SIZE_X	SIZE_Y
1	Foo	1	Foo	Foo.jpg	123	123
2	Bar	1	Bar	Bar.jpg	420	80
3	Baz	2	Baz	Baz.jpg	50	60

Mapeo de clases





No son gestionadas

- Al contrario que en EJB 2.x no son gestionadas
- Una asociación Java es unidireccional, es una referencia
- Hibernate no cambia la semántica de Java
- No es lo mismo de $A \rightarrow B$ que $B \rightarrow A$

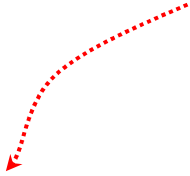


Asociaciones en Java

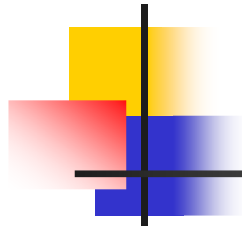
```
Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategories().add(aChild);
```

Si la relación es bidireccional
siempre hay que establecer la
relación en las dos clases

Se podría añadir un método
como este para gestionar de
forma cómoda la relación

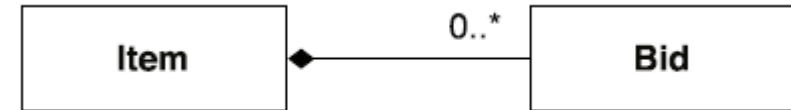


```
public void addChildCategory(Category childCategory) {
    if (childCategory == null)
        throw new IllegalArgumentException("Null child category!");
    if (childCategory.getParentCategory() != null)
        childCategory.getParentCategory().getChildCategories()
            .remove(childCategory);
    childCategory.setParentCategory(this);
    childCategories.add(childCategory);
}
```

Multiplicidad en JPA

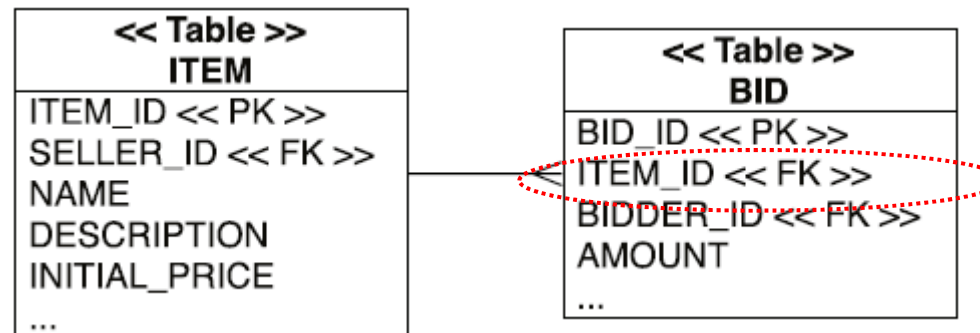
- one-to-one
- many-to-many
- one-to-many
- many-to-one
 - son direccionales, esta es la inversa de una one-to-many



Unidireccional muchos a uno

```
public class Bid {
    ...
    @ManyToOne
    @JoinColumn(nullable = false)
    private Item item;
    ...
}
```

Bid siempre debe tener un Item





Bidireccional uno a muchos

```
public class Bid {
    ...
    @ManyToOne
    @JoinColumn(nullable = false)
    private Item item;
    ...
}

public class Item {
    ...
    @OneToMany(mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();
    ...
    public void addBid(Bid bid) {
        bid.setItem(this);
        bids.add(bid);
    }
}
```

Doble actualización

```
...  
@OneToMany(mappedBy = "item")  
private Set<Bid> bids = new HashSet<Bid>();
```

```
bid.setItem(item);  
bids.add(bid);
```

La doble actualización

- En java es necesaria pero en SQL la asociación es una foreign key.
 - Solo se actualiza el campo en una tabla.
- Hibernate vigila ambos extremos y detecta las dos modificaciones en Java
- Se producirán dos INSERT o dos UPDATE (según) cuando sólo una es necesaria
- Para evitarlo se marca un extremo con `mappedBy="campo_FK_del_otro_extremo"`

Propagación en cascada

```
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 llama de las dos formas
- Se da en las relaciones padre/hijo (los hijos dependen del padre)
- Se puede especificar por separado el tipo cascada deseado para cada asociación

```
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"

nov-08



Tipos de cascada hibernate

none	Por defecto. No se propaga nada.
save-update	Propaga llamadas a <code>save()</code> , <code>update()</code> y <code>saveOrUpdate()</code> . Los objetos alcanzados pueden estar <code>transient</code> o <code>detached</code> y todos quedan <code>persistent</code> .
merge	Propaga llamadas a <code>merge()</code> . Todos los hijos <code>detached</code> son replicados como persistentes, los hijos <code>transient</code> pasan a <code>persistent</code> .
delete	Propaga llamadas a <code>delete()</code> .
lock	Propaga llamadas a <code>lock()</code> . Los hijos <code>detached</code> quedan <code>persistent</code> . El nivel de bloqueo no se propaga a los hijos.
replicate	Propaga llamadas a <code>replicate()</code> .
evict	Propaga llamadas a <code>evict()</code> .
refresh	Propaga llamadas a <code>refresh()</code> .
all	Incluye todas las opciones previas.
delete-orphan	Provoca el borrado de los objetos con solo sacarlos de la colección del padre.



Tipos de cascada JPA

- ALL
- MERGE
- PERSIST
- REFRESH
- REMOVE



Cascada delete-orphan

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

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

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

¿No será una composición?

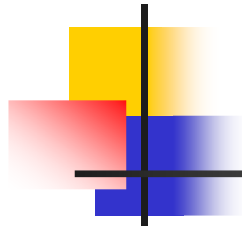
Uno o uno con foreign key

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

<< Table >> USERS	
USER_ID << PK >>	
SHIPPING_ADDRESS_ID << FK >> << UNIQUE >>	
FIRSTNAME	
LASTNAME	
USERNAME	
...	

<< Table >> ADDRESS	
ADDRESS_ID << PK >>	
STREET	
ZIPCODE	
CITY	

En la clase que no
tiene la FK



Uno a uno con la misma clave

- Dos tablas comparten la misma clave
 - Es clave primaria en las dos
 - Y además *foreign key* en una de ellas
- Problema: solo se genera una clave, la segunda tabla debe esperar a que se genere en la primera
 - Se usa un generador especial para la segunda

Uno a uno misma Clave

<< Table >> USERS	
USER_ID << PK >>	
FIRSTNAME	
LASTNAME	
USERNAME	
...	

<< Table >> ADDRESS	
ADDRESS_ID << PK >> << FK >>	
STREET	
ZIPCODE	
CITY	

Con este generador toma la clave de la otra

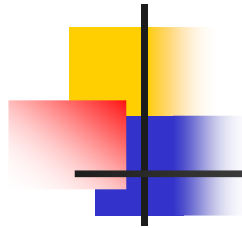
```
@Entity
public class Address {
    @Id @GeneratedValue(generator = "myForeignGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "myForeignGenerator",
        strategy = "foreign",
        parameters = @Parameter(
            name = "property", value = "user")
    )
    @Column(name = "ADDRESS_ID")
    private Long id;
    ...
    private User user;
}

public class User {
    ...
    @OneToOne
    @PrimaryKeyJoinColumn
    private Address shippingAddress;
    ...
}
```

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);
shippingAddress.setUser(newUser);

session.save(newUser);
```



Uno a muchos con Bag

- Ya está visto con SET pero se puede hacer con BAG si no se necesita ordenación y se permiten duplicados
- Al no tener que garantizar el orden ni vigilar los duplicados no hace falta cargar la colección para hacer las inserciones. Se consigue más eficiencia.



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 el orden en el que fueron insertados
- Esto es, no se ordenan después de metidos como en SortedSet (o SortedMap)

```
public class Item {  
    ...  
    @OneToMany  
    @JoinColumn(name = "ITEM_ID", nullable = false)  
    @org.hibernate.annotations.IndexColumn(name = "BID_POSITION")  
    private List<Bid> bids = new ArrayList<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

nov-08

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

... a Muchos @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>();
```

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 >>

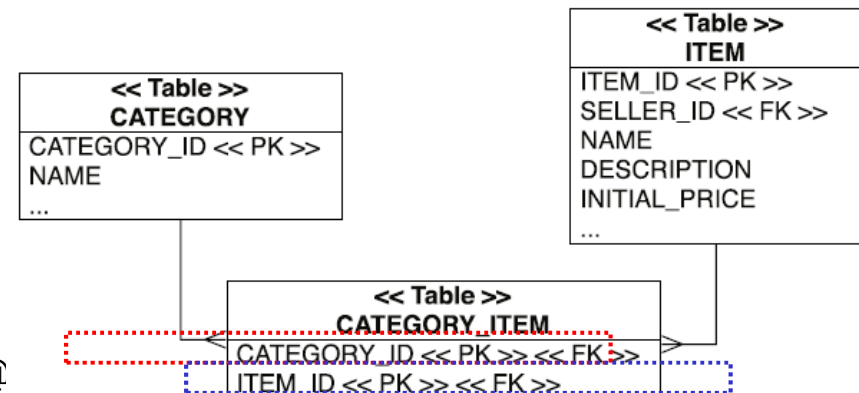
Muchos a muchos

bidireccional

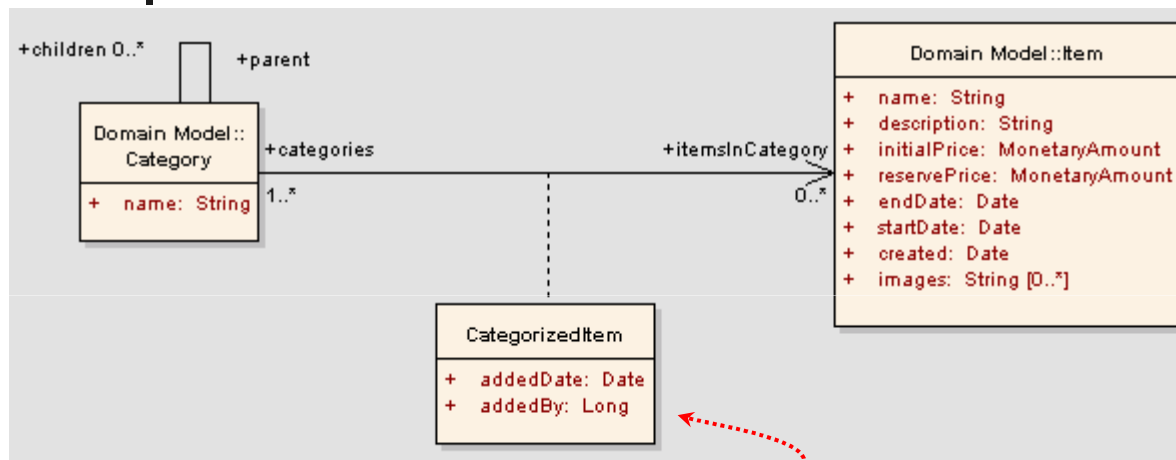
```
aCategory.getItems().add(anItem);  
anItem.getCategories().add(aCategory);
```

```
@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

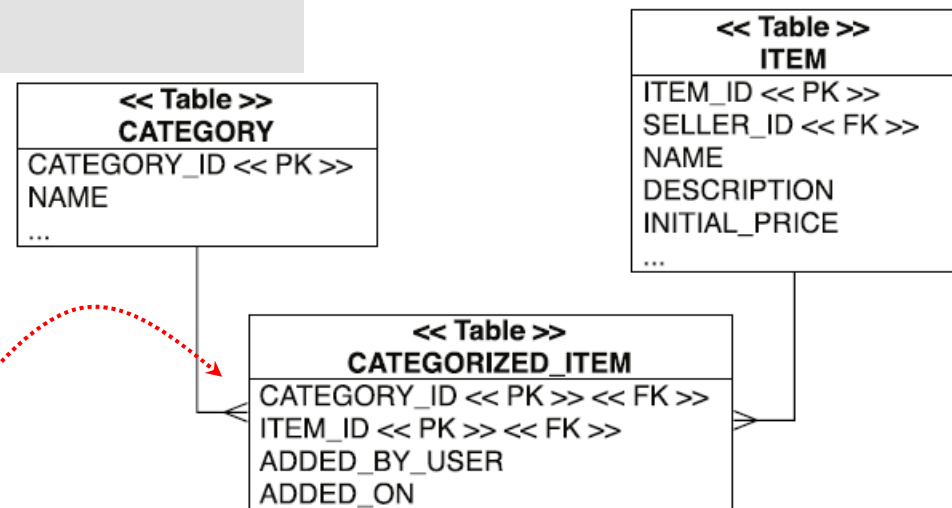


Mapeo de clases asociativas



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

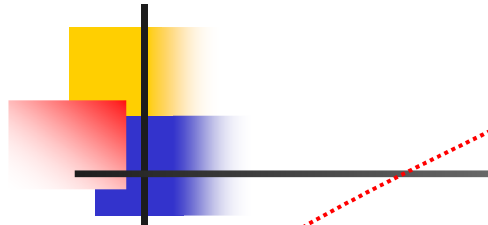
En BDD una muchos a muchos
con más columnas



Clase asociativa

```
@Entity
@Table(name = "CATEGORIZED_ITEM")
public class CategorizedItem {
    @Embeddable
    public static class Id implements Serializable {
        @Column(name = "CATEGORY_ID")
        private Long categoryId;
        @Column(name = "ITEM_ID")
        private Long itemId;
        public Id() {}
        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }
        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
                Id that = (Id)o;
                return this.categoryId.equals(that.categoryId) &&
                    this.itemId.equals(that.itemId);
            } else {
                return false;
            }
        }
        public int hashCode() {
            return categoryId.hashCode() + itemId.hashCode();
        }
    }
}
```

Clase para la clave
compuesta



Clave compuesta:
la clase Id debe cumplir
unas condiciones

```
@Entity
@Table(name = "CATEGORIZED_ITEM")
public class CategorizedItem {
    @Embeddable
    public static class Id implements Serializable {

        @EmbeddedId
        private Id id = new Id();
        @Column(name = "ADDED_BY_USER")
        private String username;
        @Column(name = "ADDED_ON")
        private Date dateAdded = new Date();
        @ManyToOne
        @JoinColumn(name="ITEM_ID", insertable = false, updatable = false)
        private Item item;
        @ManyToOne
        @JoinColumn(name="CATEGORY_ID", insertable = false, updatable = false)
        private Category category;

        public CategorizedItem() {}
        public CategorizedItem(String username, Category category, Item item) {
            // Set fields
            this.username = username;
            this.category = category;
            this.item = item;
            // Set identifier values
            this.id.categoryId = category.getId();
            this.id.itemId = item.getId();
            // Guarantee referential integrity
            category.getCategorizedItems().add(this);
            item.getCategorizedItems().add(this);
        }
        // Getter and setter methods
        ...
    }
}
```



Primary Key Class:

- Es una clase Java (POJO) pública.
- Constructor público sin argumentos.
- Si hay acceso por get/set deben ser public o protected.
- Debe ser serializable.
- Define equals() and hashCode().