

Programació



UT14.4 Herència i associacions

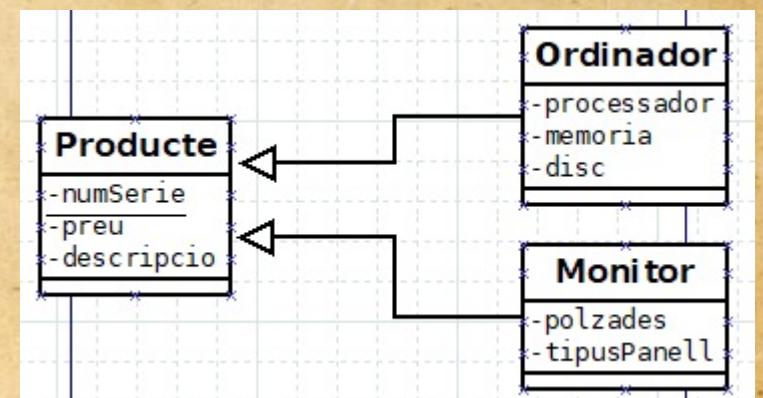
Herència

Hem vist com gestionar entitats aïllades, però això és una cosa inusual ja que en una aplicació les entitats **soLEN tINDRE ALGUN tipUS DE relaciÓ** entre elles, com pot ser l'herència o l'associació, unes de les principals eines de la POO.

Les bases de dades relacions **no entenen el concepte d'herència**, per la qual cosa usar herència en JPA pot resultar complicat. No existeix una forma única d'implementar l'herència en una base de dades, així que haurem de triar com la representarem en el model relacional.

Per a permetre optimitzar les lectures o escriptures a la base de dades existeixen tres estratègies per a mapejar l'herència:

- *SINGLE_TABLE*
- *JOINED*
- *TABLE_PER_CLASS*



Herència en taula única

Esta és l'estrategia que s'usarà per defecte si no especificuem cap altra. És la solució més senzilla i la que sol donar el millor rendiment (si podem modificar les nostres taules per a afegir una columna discriminadora). És especialment útil per a l'herència **superposada**.

En esta estrategia s'usa **una única taula per a emmagatzemar totes les instàncies de totes les classes que formen part de l'herència**. Esta taula tindrà una columna per a cada atribut de cada classe, a més d'una **columna discriminadora**, que s'usa per a determinar a quina classe correspon cadascuna de les files.

La taula de BD resultant del diagrama anterior **seria una taula única** com esta:

Producte(numSerie, preu, descripció, processador, memòria, disc, polzades,
tipusPanell, discrimina)

Herència en taula única

Classe “Producte”

```
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name="discrimina")  
@ForceDiscriminator  
@Table(name="productes")  
public abstract class Producte {  
    @Id  
    private int numSerie;  
    private double preu;  
    private String descripcio;  
    //... altres mètodes  
}
```

Subclasse “Ordinador”

```
@Entity  
@DiscriminatorValue("0")  
public class Ordinador  
extends Producte, implements Serializable {  
    private String processador;  
    private int memoria;  
    private double memoria;  
    //... altres mètodes  
}
```

Subclasse “Monitor”

```
@Entity  
@DiscriminatorValue("M")  
public class Monitor  
extends Producte, implements Serializable {  
    private int polzades;  
    private String tipusPanell;  
    //... altres mètodes  
}
```

Herència en taules enllaçades

Esta estratègia és especialment adequada per a aquells models relacionals que seguisquen l'estructura definida en el model d'objectes, per la qual cosa sol utilitzar-se quan este últim model és el que es dissenya en primer lloc. És especialment útil per a l'herència parcial i disjunta.

En esta estratègia **es defineix una taula per a cada classe, tant pare com filla**. Cada taula d'una classe **filla** **conté únicament els atributs que la diferencien** de la classe pare, i la classe pare ha de contindre a **més una columna discriminadora**, com en l'estratègia anterior. A més, cada taula ha d'emmagatzemar **la clau primària de l'objecte**, encara que només la definirem en la classe arrel. Totes les classes de la jerarquia han de compartir el mateix atribut identificador. Un conjunt de taules per a esta estratègia té el següent aspecte:

```
Producte(numSerie, preu, descripcio, discrimina)
Ordinador(numSerie, processador, memoria, disc)
Monitor(numSerie, polzades, tipusPanell)
```

Herència en taules enllaçades

Classe "Producte"

```
@Entity  
@Inheritance(strategy=InheritanceType.JOINED)  
@DiscriminatorColumn(name="discrimina")  
@Table(name="products")  
public abstract class Producte {  
    @Id  
    private int numSerie;  
    private double preu;  
    private String descripcio;  
    //... altres mètodes  
}
```

Subclasse "Ordinador"

```
@Entity  
@DiscriminatorValue("0")  
@Table(name="ordinadors")  
public class Ordinador  
extends Producte, implements Serializable {  
    private String processador;  
    private int memoria;  
    private double memoria;  
    //... altres mètodes  
}
```

Subclasse "Monitor"

```
@Entity  
@DiscriminatorValue("M")  
@Table(name="monitors")  
public class Monitor  
extends Producte, implements Serializable {  
    private int polzades;  
    private String tipusPanell;  
    //... altres mètodes  
}
```

Herència en una taula per a cada classe

Utilitzarem esta estratègia quan vulguem mantindre l'herència únicament en el model d'objectes, fent al model relacional ignorant d'eixe fet. A més, és especialment útil quan tinguem herència completa i disjunta.

En esta estratègia s'usa una classe pare abstracta i diverses classes filles. Cada **classe filla** és representada en el model relacional en forma d'una taula que **emmagatzema tots els seus atributs i tots els atributs de la seua superclasse**.

Ací podem vore un exemple de taules que segueixen esta estratègia:

Producte(numSerie, preu, descripcio)

Ordinador(numSerie, preu, descripcio, processador, memoria, disc)

Monitor(numSerie, preu, descripcio, polzades, tipusPanell)

Herència en una taula per a cada classe

Classe "Producte"

```
@Entity  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
@Table(name="productes")  
public abstract class Producte {  
    @Id  
    private int numSerie;  
    private double preu;  
    private String descripcio;  
    //... altres mètodes  
}
```

Subclasse "Ordinador"

```
@Entity  
@Table(name="ordinadors")  
public class Ordinador  
extends Producte, implements Serializable {  
    private String processador;  
    private int memoria;  
    private double memoria;  
    //... altres mètodes  
}
```

Subclasse "Monitor"

```
@Entity  
@Table(name="monitors")  
public class Monitor  
extends Producte, implements Serializable {  
    private int polzades;  
    private String tipusPanell;  
    //... altres mètodes  
}
```

Associacions

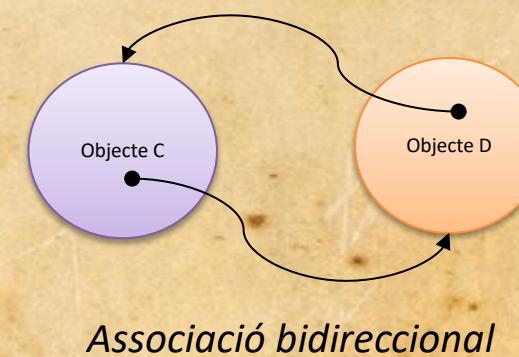
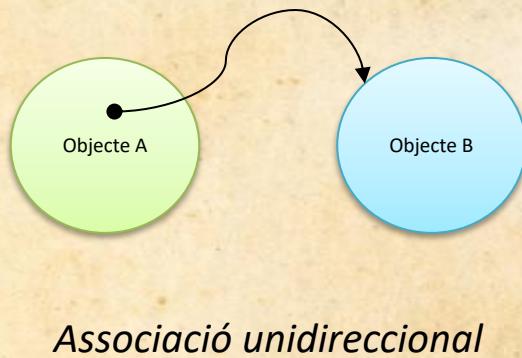
Entre les classes que componen el model de domini existeixen relacions que permeten modelar la realitat. En una base de dades, les relacions entre els registres s'implementen **mitjançant claus foranes**. No obstant això en la POO no ocorre el mateix, sinó que les associacions s'implementen **mitjançant referències**.

Si des d'un objecte dispose de la referència d'un segon objecte, des del primer podré accedir al segon (però **no està garantit el camí invers**). És important distingir que mentre que una BD les relacions són sempre bidireccionals, en les classes les associacions seran unidireccionals o bidireccionals segons la nostra implementació.

Associacions

Una associació es defineix a partir de dos característiques:

- **La navegabilitat:** el sentit de les referències (unidireccional o bidireccional)
- **La cardinalitat:** el nombre d'objectes de totes dues entitats que participen en l'associació (*un a un, un a molts, molts a un o bé molts a molts*)



Associacions. Entitat propietària

Diem que l'entitat que es correspon amb la taula que conté la clau aliena cap a l'altra, és la **propietària de la relació**.

Per a especificar les columnes que són claus alienes en una relació s'utilitzen les anotacions **@JoinColumn** i l'atribut **mappedBy**. La primera identifica l'entitat propietària de la relació i permet definir el nom de la columna que defineix la clau aliena (equivalent a l'anotació **@Column** en altres atributs). El seu ús és opcional, encara que és recomanable, ja que fa el mapatge més fàcil d'entendre. La segona s'usa en l'atribut de l'entitat no propietària que no es mapeja en cap columna de la taula. Com veurem més endavant, este element ha d'indicar el nom de l'atribut que representa la clau aliena en l'altra entitat.

També és possible mapejar una relació utilitzant una taula “join”, una taula auxiliar amb dos claus alienes cap a les taules de la relació. No ens dona temps en aquest tema d'explicar com fer-ho. Si estàs interessat pots consultar en l'especificació de JPA.

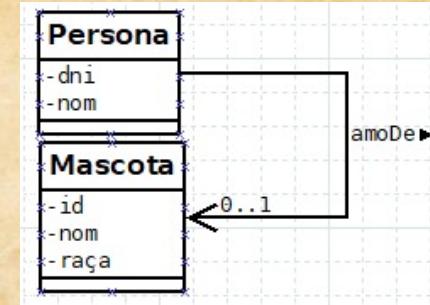
Associacions. @OneToOne unidireccional

Podem vore esta associació amb l'exemple de les persones i les seues mascotes:

- Una persona és propietària d'una única mascota.
- Una mascota té un únic amo.

```
@Entity
public class Persona{
    @Id
    private String dni;
    private String nom;
    @OneToOne
    // @JoinColumn opcional!!
    // name -> nom columna FK de la BD
    /* referencedColumnName -> nom
       columna PK de l'altra taula */
    @JoinColumn(name="mascota",
               referencedColumnName="id")
    private Mascota mascota;
    // Resta de l'implementació ...
}
```

```
@Entity
public class Mascota{
    @Id
    private int id;
    private String nom;
    private String rassa;
    // Resta de la implementació
}
```



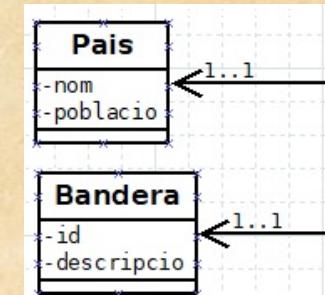
Atributs de l'anotació @OneToOne

Associacions. @OneToOne bidireccional

Podem vore esta associació amb la relació entre un país i la seu bandera.

- *Un país té una única bandera .*
- *Una bandera només pertany a un país.*

```
@Entity
public class Pais{
    @Id
    private String nom;
    private Long poblacio;
    /* @OneToOne sense mappedBy
       indica que és l'entitat
       propietària de l'associació */
    @OneToOne
    @JoinColumn(name="bandera",
               referencedColumnName="id")
    private Bandera bandera;
    // Resta de la implementació ...
}
```



```
@Entity
public class Bandera{
    @Id
    private int id;
    private String descripcio;
    /* mappedBy indica l'atribut
       relacionat de l'altra entitat, i
       permet navegació bidireccional */
    @OneToOne(mappedBy = "bandera")
    private Pais pais;
    // Resta de la implementació
}
```

Associacions. @OneToOne bidireccional

Cal fer notar que **és molt important actualitzar el costat de l'associació que és el propietari de la mateixa** i que conté la clau aliena. Si ho férem a l'inrevés i actualitzàrem l'atribut "pais" de Bandera tindríem la desagradable sorpresa que la relació no s'actualitza en la base de dades. Per això seria incorrecte fer el següent:

```
em.getTransaction().begin();
Pais p1 = new Pais("Itàlia", 60317000);
Bandera b1 = new Bandera(25, "Tricolor verd-blanc-roig");
b1.setPais(p1);
em.getTransaction().commit();
```

Seria incorrecte perquè estem actualitzant el costat invers de la relació, no el costat propietari. Encara que la relació és bidireccional, l'actualització només es pot fer en el costat del propietari de la relació. Per a evitar possibles errors, el millor és implementar únicament el mètode set en el costat del propietari de la relació i en l'altre costat (en l'entitat Emprat) posar com a private l'atribut de la relació i no definir cap mètode set.

Associacions. @OneToMany unidireccional

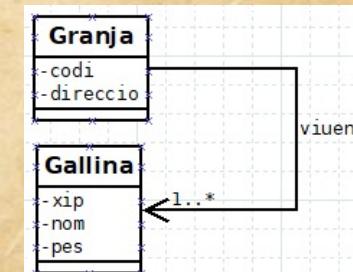
JPA admet associacions “un a molts” amb atributs de tipus List, Map o Set

Podem vore esta associació amb l'exemple de una granja i les seues gallines:

- *En una granja viuen una o més gallines.*
- *Una gallina pertany a una i només una granja.*

```
@Entity
public class Granja{
    @Id
    private int codi;
    private String direccio;
    @OneToMany
    /* No indiquem @JoinColumn
       ja que JPA genera una taula
       Granja_Gallina controlada */
    private List<Gallina> gallines;
    // Resta de la implementació ...
}
```

```
@Entity
public class Gallina{
    @Id
    private int xip;
    private String nom;
    private int pes;
    // Resta de la implementació
}
```



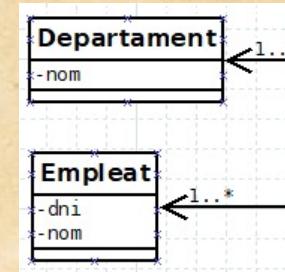
Per a poder gestionar l'associació cal que la classe **Granja** instàncie alguna de les classes que implementen l'interfície **List** (per exemple `this.gallines = new LinkedList<>();`). També cal que la classe gestione la col·lecció amb mètodes per afegir, modificar o eliminar gallines de la llista.

Associacions. @OneToMany bidireccional

Utilitzarem com a exemple la relació entre els departaments i empleats d'una empresa:

- *En un departament treballen molts empleats.*
- *Els empleats només fan el seu treball en un departament.*

```
@Entity  
public class Empleat{  
    @Id  
    private String dni;  
    private String nom;  
    @ManyToOne  
    /* En este cas no genera una  
       taula intermèdia i podem  
       definir @JoinColumn      */  
    @JoinColumn(name="departament",  
               referencedColumnName="nom")  
    private Departament dpt;  
    // Resta de la implementació ...  
}
```



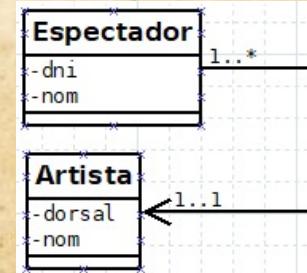
```
@Entity  
public class Departament{  
    @Id  
    private String nom;  
    @OneToMany(mappedBy = "dpt")  
    List<Empleat> empleats;  
    // Resta de la implementació  
}
```

- Una associació **@OneToMany** és una associació **@ManyToOne** en sentit contrari.
- L'anotació **@ManyToOne** NO admet **mappedBy**, per això sempre anirà en el **@OneToMany**

Associacions. @ManyToOne unidireccional

Com a exemple modelarem un concurs de talents on:

- *Cada espectador vota al seu artista favorit.*
- *Un artista pot ser votat per molts espectadors.*



```
@Entity  
public class Espectador{  
    @Id  
    private String dni;  
    private String nom;  
    @ManyToOne  
    @JoinColumn(name="artista",  
               referencedColumnName="dorsal")  
    private Artista artistaVotat;  
    // Resta de la implementació  
}
```

```
@Entity  
public class Artista{  
    @Id  
    private int dorsal;  
    private String nom;  
    // Resta de la implementació  
}
```

Associacions. @ManyToOne bidireccional

Per a l'exemple anterior seria tan senzill com afegir l'anotació **@OneToMany** en la classe Artista de la següent manera:

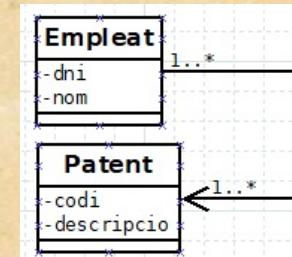
```
@Entity
public class Artista{
    @Id
    private int dorsal;
    private String nom;
    @OneToMany(mappedBy = "artistaVotat")
    List<Espectador> espectadorsLiVoten;
    // Resta de la implementació
}
```

Associacions. @ManyToMany unidireccional

Podem vore esta associació amb empleats que treballen en patents:

- *Un empleat pot treballar en moltes patents.*
- *Una patent pot ser desenvolupada per més d'un empleat.*

```
@Entity  
public class Empleat{  
    @Id  
    private String dni;  
    private String nom;  
    @ManyToMany  
    private Set<Patent> patents;  
    public Empleat(){  
        this.patents = new HashSet<>();  
    }  
    public Set<Patent> getPatents(){  
        return this.patents;  
    }  
    public void setPatents(Set<Patent> patents){  
        this.patents = patents;  
    }  
    // Resta de la implementació
```

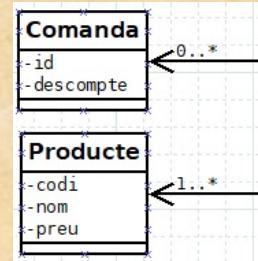


```
@Entity  
public class Patent{  
    @Id  
    private int codi;  
    private String descripcio;  
    // Resta de la implementació  
}
```

Associacions. @ManyToMany bidireccional

Podem vore esta associació amb comandes que contenen productes:

- Una comanda està composta per diversos productes.
- Un producte pot estar present en diverses comandes.



```
@Entity  
public class Comanda{  
    @Id  
    private int id;  
    private int descompte;  
    @ManyToMany  
    private Set<Producte> productes;  
    public Comanda(){  
        this.productes = new HashSet<>();  
    }  
    public Set<Producte> getProducts(){  
        return this.productes;  
    }  
    public void setProducts(Set<Producte> productes){  
        this.productes = productes;  
    }  
    // Resta de la implementació  
}
```

```
@Entity  
public class Producte{  
    @Id  
    private int codi;  
    private String nom;  
    private double preu;  
    @ManyToMany(mappedBy = "productes")  
    private Set<Comanda> comandes;  
    public Producte(){  
        this.comandes = new HashSet<>();  
    }  
    public Set<Comanda> getComandes(){  
        return this.comandes;  
    }  
    public void setComandes(Set<Comanda> comandes){  
        this.comandes = comandes;  
    }  
    // Resta de la implementació  
}
```

Associacions. Taula intermèdia.

Igual com passa a l'associació **@OneToMany** (unidireccional) amb les associacions **@ManyToMany** es genera una **taula intermèdia** en la BD que té com a clau primària la combinació dels dos camps que seran també clau aliena a les respectives taules de les entitats.

En el cas de ja existir la taula “join”, o de necessitar un nom específic per als seus elements, podem utilitzar l'anotació **@JoinTable** de la següent forma:

```
public class Comanda{  
    @Id  
    private int id;  
    private int descompte;  
    @ManyToMany  
    @JoinTable(  
        name = "linies_comanda",      // Nom de la taula intermèdia  
        joinColumns = @JoinColumn(name = "comanda"),    // nom de la columna FK a la taula d'aquesta entitat  
        inverseJoinColumns = @JoinColumn(name="producte") // nom de la columna FK a l'altra taula de la relació  
    private Set<Producte> productes;  
    // Resta de la implementació  
}
```

Associacions. Columnes addicionals en la taula intermèdia.

És bastant usual trobar columnes addicionals en les taules “join”, a més de les dues columnes alienes.

Imagina en el cas anterior, que a més de guardar la clau aliena a la comanda i al producte volem guardar la quantitat de cada línia:

Comandes(id, descompte)

Productes(codi, nom, preu)

Linies(comanda, producte, quantitat)

Este problema és complex de solucionar, i existeixen diverses formes per a resoldre'l. Comentarem una d'elles.

Associacions. Columnes addicionals en la taula intermèdia.

```
@Entity  
public class LiniaPK  
implements Serializable{  
    @Column(name = "comanda")  
    private int idComanda;  
    @Column(name = "producte")  
    private int codiProducte;  
  
    // Constructor, getters i setters...  
    /* El mètode "equals" i "hashCode" ha  
     * de tindre en compte els dos atributs */  
}
```

Primer de tot hem de crear una classe (no serà una entitat) encastable, composada pels atributs que conformen la clau primaria de la taula intermèdia o “join table”

Associacions. Columnes addicionals en la taula intermèdia.

```
@Entity
public class Linia {
    @EmbeddedId
    private LiniaPK id;
    @ManyToOne
    @MapsId("idComanda")
    private Comanda comanda;
    @ManyToOne
    @MapsId("codiProducte")
    private Producte producte;
    private int quantitat;
    public Linia(){};
    public Linia(Comanda comanda, Producte producte, int quantitat){
        this.comanda = comanda;
        this.producte = producte;
        this.quantitat = quantitat;
        this.id = new LiniaPK(comanda.getId(), producte.getCodi());
    }
    // getters i setters...
    /* El mètode "equals" i "hashCode" ha
       de tindre en compte "comanda" i "producte" */
}
```

Després hem de mapejar la taula intermèdia utilitzant una entitat

@EmbeddedId indica que la classe “LiniaPK” conté els atributs que conformen la clau primaria de “Linia”.

@MapsId indica el nom de l’atribut de la classe encastrada (embedded) que proporciona el mapejat.

Associacions. Columnes addicionals en la taula intermèdia.

```
@Entity
public class Comanda{
    @Id
    private int id;
    private int descompte;
    @OneToMany(mappedBy = "comanda")
    private Set<Linia> linies;
    public Comanda(){
        this.linies = new HashSet<>();
    }
    // Resta de la implementació...
    /* Recorda els mètodes per afegir, modificar
       o eliminar línies de la col·lecció "linies" */
}
```

L'entitat Comanda maparà el costat @OneToMany per a l'atribut «comanda» a l'entitat d'unió Linia

Associacions. Columnes addicionals en la taula intermèdia.

```
@Entity  
public class Producte{  
    @Id  
    private int codi;  
    private String nom;  
    private double preu;  
    @OneToMany(mappedBy = "producte")  
    private Set<Linia> linies;  
    public Producte(){  
        this.linies = new HashSet<>();  
    }  
    // Resta de la implementació...  
}
```

L'entitat Producte maparà el costat @OneToMany per a l'atribut «producte» a l'entitat d'unió Linia

} Estes línies de codi són opcionals. Pot donar-se el cas que no vulguem guardar les linies en l'entitat Producte.

Associacions. Configuració.

A més del paràmetre “mappedBy” podem configurar les associacions mitjançant alguns paràmetres opcionals a les anotacions @OneToOne, @OneToMany, @ManyToOne y @ManyToMany; per exemple disposem de fetch, cascade, optional i orphanRemoval.

No totes les anotacions admeten estos paràmetres, tenim:

	<i>fetch</i>	<i>cascade</i>	<i>mappedBy</i>	<i>optional</i>	<i>orphanRemoval</i>
@OneToOne	✓	✓	✓	✓	✓
@OneToMany	✓	✓	✓		✓
@ManyToOne	✓	✓		✓	
@ManyToMany	✓	✓	✓		

Associacions. Configuració. Fetch

El paràmetre “fetch” defineix quan JPA obté les entitats relacionades de la base de dades i és un dels elements crucials per a un nivell de persistència ràpid i eficient. Té dos possibles valors:

- **FetchType.EAGER** diu a JPA que obtinga tots els elements d'una relació quan selecciona l'entitat arrel.
- **FetchType.LAZY** diu a JPA que només recupere les entitats relacionades de la base de dades quan s'utilitze la relació. Esta és una bona idea en general perquè no hi ha cap motiu per seleccionar entitats que no necessitem per a cada ús.

Per defecte, dels quatre tipus de relació entre entitats permesos en JPA només dos són per defecte de tipus Lazy: @OneToMany i @ManyToMany. Això té molt sentit, perquè són estos dos tipus els que a més objectes connecten en l'altre costat de la relació. Com que és el comportament per defecte, no hem de fer res per a declarar estos relacionaments com Lazy (és el seu comportament implícit).

```
@Entity
public class Treballador{
    // ...
    @OneToOne(fetch = "FetchType.LAZY")
    private Domicili domicili;
    // Quan seleccionem per exemple una llista de treballadors, no consultarem el domicili de tots
    // Només quan accedim al mètode "getDomicili()" d'un treballador, farà la consulta.
}
```

Associacions. Configuració. Cascade

Una operació en cascada és aquella que quan l'apliquem a un objecte es propaga de forma automàtica als objectes amb els quals té una associació.

Per exemple en l'exemple de la granja i les gallines, podem configurar l'operació de persistència en cascada, el que implica que si persistim un objecte de tipus “Granja” es persistiran de forma automàtica tots els objectes de tipus “Gallina” que conté, sense necessitat d'executar per a cadascun d'ells el mètode “persist”. Igualment podem aplicar l'esborrat en cascada, de manera que si eliminem una granja també eliminarem totes les seues gallines.

El paràmetre pot tindre els següents valors:

- **CascadeType.PERSIST**: Si persistim un objecte, es persistiran de forma automàtica tots els objectes amb els quals mantinga una relació.
- **CascadeType.REMOVE**: Si eliminem un objecte, s'eliminaran de forma automàtica tots els objectes amb els quals mantinga una relació.
- **Ídem per a DETACH, MERGE, i REFRESH.**
- **CascadeType.ALL**: Combina els anteriors.

```
@Entity  
public class Granja  
implements Serializable {  
    //...  
    @OneToMany(cascade = {CascadeType.ALL})  
    List<Gallina> gallines;  
    //...  
}
```

Associacions. Configuració. Optional

Podem definir si la relació es opcional o no, es a dir, si permetem que l'atribut que crea l'associació puga ser **null**.

Si tornem al nostre exemple de les persones i les mascotes, i decidim que totes les persones han de tindre una mascota relacionada, hem d'especificar que la relació no és opcional:

```
@Entity  
public class Persona  
implements Serializable{  
    // ...  
    @OneToOne(optional = false)  
    private Mascota mascota;  
    // ...  
}
```

D'esta manera l'entitat **Persona** no admet valors nuls en el seu atribut, i si li assignem un valor nul llançarà una excepció.

Associacions. Configuració. OrphanRemoval

Un objecte orfe és aquell que ha sigut disconnectat d'una relació. Esta desconexió pot ocórrer per dos motius: hem eliminat l'objecte que forma l'altra part de la relació o, simplement, en l'objecte que forma l'altra part de la relació s'ha modificat la referència. Si recordem l'exemple dels empleats i els departaments podríem fer:

```
@Entity  
public class Empleat{  
    @Id  
    private String dni;  
    private String nom;  
    @ManyToOne  
    /* En este cas no genera una  
       taula intermèdia i podem  
       definir @JoinColumn */  
    @JoinColumn(name="departament",  
               referencedColumnName="nom")  
    private Departament dpt;  
    // Resta de la implementació ...  
}
```

```
@Entity  
public class Departament  
implements Serializable{  
    // ...  
    @OneToMany(mappedBy = "dpt", orphanRemoval = true)  
    private List<Empleat> empleats;  
    // ...  
}
```

De tal manera que:

```
// Suposant que "empleat1" i "departament1"  
// són entitats gestionades (managed)  
em.getTransaction().begin();  
empleat1.setDepartament(null);  
em.getTransaction().commit();  
// Provoca que s'elimine l'empleat de la BD
```

```
// Suposant que "empleat1" i "departament1"  
// són entitats gestionades (managed)  
em.getTransaction().begin();  
departament1.removeEmpleat(empleat1);  
em.getTransaction().commit();  
// Provoca que s'elimine l'empleat de la BD
```