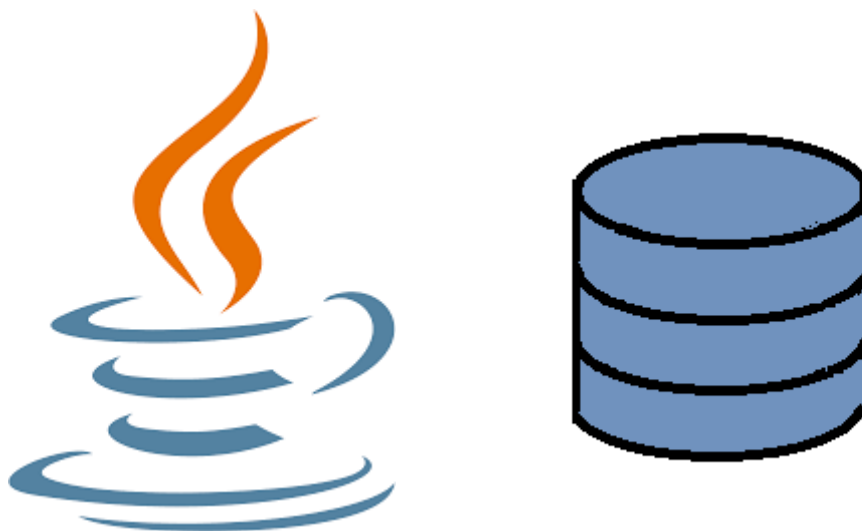


CFGS Desenvolupament d'Aplicacions Multiplataforma (DAM)

Mòdul 6 – Accés a dades

UF2 – Persistència en BDR-BDOR-BDOO

Iniciació a JPA



Índex de contingut

1. Introducció a JPA.....	3
2. Marcant les classes persistents i les seves claus amb anotacions.....	3
3. Gestionant la connexió amb la base de dades.....	5
a) EntityManager.....	5
b) Tancar.....	5
c) Transaccions.....	5
4. Realitzant operacions sobre objectes individuals.....	6
a) Operacions disponibles.....	6
b) Exemples.....	7
5. Definint amb anotacions les característiques del mapatge objecte-relacional.....	9
a) Nom de la taula que representarà els objectes de la classe:.....	9
b) Nom de la taula que representarà una relació molts a molts:.....	10
c) Característiques dels camps:.....	10
d) Implementació de l'herència:.....	11
6. Definint relacions entre classes persistents i les seves claus amb anotacions.....	15
a) Tipus de relacions:.....	15
b) Relacions 1 a 1 navegables en un sentit.....	16
c) Relacions 1 a 1 navegables en tots dos sentits.....	18
d) Relacions 1 a N navegables d'N a 1.....	19
e) Relacions 1 a N navegables en tots dos sentits.....	21
f) Relacions M a N navegables en un sol sentit.....	22
g) Relacions M a N navegables en tots dos sentits.....	23
7. JPQL. Definició de consultes amb nom amb anotacions. JPQL encastat.....	25
a) Introducció a JPQL.....	25
b) Definició de consultes amb nom.....	26
c) JPQL encastat.....	27
8. Especificació de la persistència sense anotacions.....	29
a) Una entitat.....	30
b) La implementació de l'herència.....	31
c) Consultes amb nom (<i>named queries</i>).....	31
d) Camps i relacions.....	32

1. Introducció a JPA

JPA (Java Persistence API) és una API (Application Programming Interface) que proporciona mecanismes per fer persistents els objectes que són a memòria d'un programa escrit en Java. Lògicament, també permet recuperar-los posteriorment.

Està dissenyada per realitzar el mapatge objecte-relacional. És a dir, per permetre al programa fer persistents i recuperar els objectes directament d'una base de dades relacional, sense haver de tenir en compte que, en realitat, està treballant amb taules i no amb objectes. Malgrat això, com les operacions que proporciona estan orientades a objectes, també és utilitzada per la base de dades orientada objecte ObjectDB.

Aquesta API està formada per una sèrie d'interfícies, classes i anotacions*. La seva implementació és una biblioteca de Java (és a dir, un fitxer .jar) que conté una de les implementacions possibles d'aquestes interfícies, classes i anotacions.

Les anotacions serveixen per:

- Indicar quines classes poden tenir objectes persistents.
- Quina serà la clau.
- Definir les relacions entre les classes.
- Definir consultes amb nom. Es defineixen amb el llenguatge JPQL.
- Definir característiques del mapatge entre objectes per una banda i, per l'altra, taules i camps.

Les classes i interfícies proporcionen mètodes per:

- Gestionar la connexió amb la base de dades.
- Realitzar consultes, altes, esborrats i modificacions d'objectes individuals.
- Realitzar consultes avançades amb el llenguatge JPQL.

JPQL és un llenguatge de sintaxi similar a la de SQL, però que treballa amb objectes de la base de dades. Així, en el cas de mapatge objecte-relacional fa transparents al programa les taules i els camps que representen aquests objectes. Aquesta característica proporciona a JPQL molta potència. Per exemple, no cal utilitzar *joins* per navegar a través de les relacions entre les classes.

En els següents apartats, suposarem que s'han realitzat els passos per treballar amb JPA indicats al document **BDOO i mapatge O-R** que teniu a l'aula.

2. Marcant les classes persistents i les seves claus amb anotacions

Davant de cada classe que vulguem fer persistent caldrà posar l'anotació **@Entity**. A més, per indicar quina és la clau cal afegir-hi **@Id**. Aquesta anotació pot afegir-se tant davant de la dada com davant del seu *getter*. La ubicació indica a JPA si ha d'utilitzar directament la dada o accedir-hi a través dels *getters* i *setters*. Nosaltres el posarem davant del *getter*.

Com exemple, treballarem amb la classe *Assignatura*, que representa una assignatura d'un pla d'estudis. Tindrà codi, nom i número d'hores com atributs. El codi serà l'atribut clau que identificarà cada assignatura.

La codificació seria la que segueix.

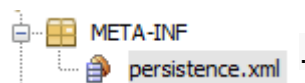
*Les anotacions són unes directives que no tenen cap efecte directe sobre l'execució del programa i que s'insereixen en el codi i poden ser consultades per instruccions del propi codi en temps d'execució,

Classe a fer persistent	Classe amb les anotacions
<pre> package exemple; public class Assignatura implements Serializable{ private String codi; private String nom; private int hores; public String getCodi() { return codi; } public void setCodi(String codi){ this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom){ this.nom = nom; } public int getHores() { return hores; } public void setHores(int hores){ this.hores = hores; } } </pre>	<pre> package exemple; @Entity public class Assignatura implements Serializable{ private String codi; private String nom; private int hores; @Id public String getCodi() { return codi; } public void setCodi(String codi){ this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom){ this.nom = nom; } public int getHores() { return hores; } public void setHores(int hores){ this.hores = hores; } } </pre>

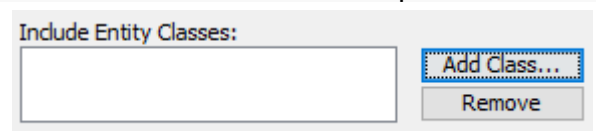
Els *import* no s'han inclòs perquè el propi entorn dona facilitats per incloure'l.

A continuació, cal indicar a la unitat de persistència que aquesta classe serà persistent de la següent manera:

Cal obrir el fitxer META-INF/persistence.xml



S'obrirà la finestra d'edició d'aquest fitxer. En ella, cal fer clic al botó *Add Class...* per seleccionar, en la llista que ens apareixerà, les classes que hem assenyalat com a persistents. Un cop fet, ens apareixeran al quadre de text titulat *Include Entity Classes:*



3. Gestionant la connexió amb la base de dades

a) EntityManager

Per gestionar la connexió amb la base de dades, en primer lloc cal tenir un objecte que implementi la interfície *EntityManager*. Aquesta permet interaccionar amb els objectes que es fan persistents o es recuperen de la base de dades. Aquests objectes formen el context de persistència. Aquest objecte es crea utilitzant el patró factoria.

Patró factoria

El patró factoria consisteix en crear objectes d'una classe sense cridant a mètodes d'una altra, en lloc d'utilitzar el seu constructor. Sol fer-se quan la creació d'objectes directament amb el constructor té una complexitat elevada.

Per crear un objecte *EntityManager* també s'utilitza el *patró factoria*.

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory(unitatDePersistencia);  
EntityManager em=emf.createEntityManager();
```

El paràmetre *UnitatDePersistencia* és un String amb el nom que s'ha donat a la unitat de persistència.

Un cop fet això, ja podem treballar amb l'objecte referenciat per la variable *em*. No cal obrir-lo explícitament.

b) Tancar

Podem tancar tant l'*EntityManager* com l'*EntityManagerFactory* invocant als seus mètodes *close()* respectivament. Cal tenir present, però, que si tanquem un *EntityManagerFactory* tots els *EntityManager* creades amb ella es tancaran.

c) Transaccions

Alguns mètodes que actualitzen la base de dades necessiten que hi hagi una transacció activa.

Suposem que la variable *em* referencia un *EntityManager* obert. Les següents instruccions treballen amb transaccions:

<code>em.getTransaction().begin();</code>	Inicia una transacció.
<code>em.getTransaction().commit();</code>	Finalitza una transacció confirmant els canvis realitzats.
<code>em.getTransaction().rollback();</code>	Finalitza una transacció rebutjant els canvis realitzats.

4. Realitzant operacions sobre objectes individuals

a) Operacions disponibles

Les operacions que permeten gravar i recuperar un objecte de la base de dades treballen amb el concepte d'**entorn de persistència**. Quan un objecte de memòria és dins d'aquest entorn significa que està representat també en la base de dades i que hi ha una vinculació entre l'objecte de memòria i l'objecte de la base de dades. Aquesta vinculació fa que quan es confirmi una transacció (crida `.getTransaction().commit()`) els canvis realitzats sobre l'objecte de memòria es repercuteixin també sobre l'objecte de la base de dades, inclús els canvis realitzats abans d'iniciar la transacció.

Les operacions principals sobre objectes individuals es recullen en la següent taula. En tots els casos s'assumeix que la variable *em* es refereix a un *EntityManager* obert.

Operació	Comentaris
<code>em.find(classe, valorClau)</code>	Retorna l'objecte de la base de dades que té <i>valorClau</i> com a clau i és de la classe indicada pel primer paràmetre. Aquest objecte passa a pertànyer al context de persistència. Si a la base de dades no hi ha cap objecte amb la clau indicada, retorna <i>null</i> .
<code>em.persist(objecte)</code>	Grava l'objecte i els objectes que referencia (si no hi existien) a la base de dades i l'afegeix al context de persistència. No retorna res. Requereix una transacció activa. L'objecte no ha d'existir prèviament al context de persistència; en cas contrari, llença una excepció <i>EntityExistsException</i> .(*)
<code>em.merge(objecte)</code>	Crea i retorna un nou objecte, que és còpia del paràmetre i que pertany al context de persistència. Si aquest objecte ja existia a la base de dades, s'actualitza amb les dades del paràmetre; si no existia, s'afegeix a la base de dades. També s'afegeixen els objectes que referencia , si no hi existien,. L'objecte passat com a paràmetre, si no hi era, no s'inclou al context de persistència. Requereix una transacció activa.
<code>em.remove(objecte)</code>	Esborra l'objecte de la base de dades i del context de persistència. Requereix una transacció oberta i que el paràmetre pertanyi al context de persistència.
<code>em.detach(objecte)</code>	Treu l'objecte del context de persistència si hi era. No s'esborra de la base de dades. En fer un commit no s'actualitzarà la base de dades amb el seu estat.
<code>em.refresh(objecte)</code>	Requereix que l'objecte sigui al context de persistència. Copia l'estat de la base de dades sobre l'objecte de memòria.

(*) Malauradament, a la pràctica el criteri per decidir si l'objecte és o no al context de persistència varia d'una implementació a una altra. Per exemple, el criteri pot ser mirar només la clau o mirar si l'objecte de memòria és el mateix.

b) Exemples

A tots ells suposem que l'EntityManager és obert i, en acabar, es tanca*. També suposem que *em* és un objecte de la classe *EntityManager* (igual que passa a l'EAC). La línia amb fons groc és la que conté la crida que es demostra a cada exemple.

persist	merge
Dona d'alta l'assignatura amb codi 11. Si ja existeix, es produeix un error (recordeu que implementacions diferents comproven si existeix de manera diferent).	Si existeix una assignatura amb codi 11, la modifica amb els valors nom="Matematiques" i hores=2000. Si no existeix, la dona d'alta a la base de dades, també amb aquests valors.
Assignatura a= new Assignatura(); a.setCodi(11); a.setNom("Mates"); a.setHores(200); em.getTransaction().begin(); em.persist(a); em.getTransaction().commit();	Assignatura a= new Assignatura(); a.setCodi(11); a.setNom("Matematiques"); a.setHores(2000); em.getTransaction().begin(); em.merge(a); em.getTransaction().commit();

find
Recupera de la base de dades l'assignatura amb codi 11, si aquesta existeix a la base de dades.
Assignatura a; a=em.find(Assignatura.class, 11); if(a!=null){ System.out.println("L'assignatura amb codi 11 es diu "+a.getNom()); }else{ System.out.println("L'assignatura amb codi 11 no existeix"); }

*Que sigui obert vol dir que abans s'han executat les instruccions:

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory(unitatDePersistencia);
EntityManager em=emf.createEntityManager();
```

Que es tanqui al final vol dir que s'executen les instruccions:

```
em.close();
emf.close();
```

find i commit

Incrementa en 100 el nombre d'hores de l'assignatura amb codi 11, si aquesta existeix a la base de dades.

```
Assignatura a=em.find(Assignatura.class, 11);

if(a!=null){
    a.setHores(a.getHores()+100);
    em.getTransaction().begin();
    em.getTransaction().commit();
}else{
    System.out.println("L'assignatura amb codi 11 no existeix");
}
```

detach

A diferència de l'anterior, en treure del context de persistència l'objecte *a* abans de modificar-lo amb *detach*, l'objecte de la base de dades **no** varia.

```
Assignatura a=em.find(Assignatura.class, 11);
if(a!=null){
    em.detach(a);
    a.setHores(a.getHores()+100);
    em.getTransaction().begin();
    em.getTransaction().commit();
}else{
    System.out.println("L'assignatura amb codi 11 no existeix");
}
```

remove

Esborra de la base de dades l'assignatura amb codi 11, si aquesta existeix a la base de dades

```
Assignatura a=em.find(Assignatura.class, 11);

if(a!=null){
    em.getTransaction().begin();
    em.remove(a);
    em.getTransaction().commit();
}else{
    System.out.println("L'assignatura amb codi 11 no existeix");
}
```


refresh

Malgrat que a l'objecte *a* se li modifiquen les hores, en fer el *refresh* abans de gravar-lo, recupera el valor que tenia a la base de dades

```
Assignatura a=em.find(Assignatura.class, 11);

if(a!=null){
    System.out.println("Hores:"+a.getHores());
    a.setHores(a.getHores()+100);
    em.refresh(a);
    System.out.println("Hores:"+a.getHores()+ " s'ha recuperat el valor inicial");
} else{
    System.out.println("L'assignatura amb codi 11 no existeix");
}
```

5. Definint amb anotacions les característiques del mapatge objecte-relacional

JPA està pensat per realitzar un mapatge entre les classes persistents (o entitats) i les taules on s'emmagatzemaran els seus objectes en una base de dades relacional. Quan es treballa amb una base de dades relacional sol interessar definir algunes de les característiques de la taula o dels camps per motius d'eficiència o de compatibilitat amb altres aplicacions. JPA preveu anotacions que permeten fer-ho. Quan es treballa amb ObjectDB, aquestes anotacions són **ignorades** (ja que ObjectDB no treballa amb taules).

Algunes d'aquestes anotacions són:

a) Nom de la taula que representarà els objectes de la classe:

L'anotació és **@Table(name="nomDeLaTaula")**. Va a sota de l'anotació **@Entity**.

Exemple:

```
@Entity
@Table(name="materia")
public class Assignatura implements Serializable{

    .....

}
```

Representa les assignatures en una taula anomenada *materia*.

Posar el nom de la taula en minúscules us pot evitar algun problema, ja que l'estàndard SQL (i PostgreSQL) indica que tot identificador que no vagi entre cometes dobles serà passat a minúscules.

b) Nom de la taula que representarà una relació molts a molts:

L'anotació és **@JoinTable(name="nomDeLaTaula")**. Va a sota de l'anotació **@ManyToMany**, sense la part *mappedBy* (l'anotació **@ManyToMany** es tracta als apartats [6.f](#) i [6.g](#)).

Exemple:

```
@ManyToMany
@JoinTable(name="relacionats")
public List<L> getRelacionats() {
    return relacionats;
}
```

c) Característiques dels camps:

En els camps també es pot definir el nom; a més, sobre tot en les cadenes, sol interessar definir-ne també la mida màxima (si no ho fem, tindrien una mida màxima de 255). Això es fa amb l'anotació **@Column(name="nomColumna", length=mida)**. Tant la dada *name* com la dada *length* són opcionals. A més, es poden posar altres propietats dins del parèntesis, com consta a la [documentació oficial](#). Normalment, si utilitzem aquesta anotació, almenys posarem el parèntesis amb algun element. L'anotació pot posar-se davant de la declaració de la dada o davant del *getter*. Es pot triar qualsevol dels dos llocs, però cal ser coherent amb el que s'ha decidit per l'anotació **@Id**. Al següent exemple s'ha posat davant del *getter*.

Exemple:

```
@Id
@Column(name="identificador")
public String getCodi() {
    return codi;
}

.....

@Column(name="materia_nom", length=30)
public String getNom() {
    return nom;
}
```

Representa la dada *codi* en un camp de la base de dades que s'anomenarà *identificador* i la dada *nom* en un camp que s'anomenarà *materia_nom* i tindrà una mida màxima de 30 caràcters.

Posar el nom del camp en minúscules us pot evitar algun problema, ja que l'estàndard SQL (i PostgreSQL) indica que tot identificador que no vagi entre cometes dobles serà passat a minúscules.

d) Implementació de l'herència:

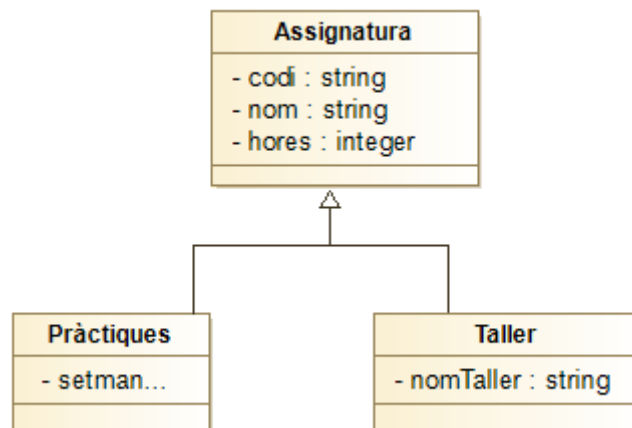
L'anotació és **@Inheritance(strategy=estratègia)**. L'estratègia pot ser una d'aquestes tres:

- **JOINED**: cada classe que hi intervé representa els objectes en una taula pròpia; la classe base utilitzarà una taula amb un camp per a cadascun dels seus atributs i un altre que indica la subclasse de l'objecte que està representant; cada subclasse n'utilitzarà una altra amb un camp per a l'identificador i, a més, un camp addicional per a cadascun dels atributs específics.
- **SINGLE_TABLE**: és l'estratègia per defecte; els objectes de totes les classes que hi intervenen es representen en una única taula que té un camp per a cada dada comuna, un altre que indica el tipus d'objecte que està representant la fila de la taula i, a més, un camp per a cada dada d'alguna de les subclasses.
- **TABLE_PER_CLASS**: cada classe que hi intervé té associada una taula amb un camp per a cada dada de l'objecte, tant per les dades heretades com per les específiques.

Si l'estratègia és **JOINED** o **SINGLE_TABLE**, com s'ha dit, es poden indicar més característiques (totes són **opcionals**):

- **@DiscriminatorColumn(name="nomColumna", discriminatorType=tipus, length=mida)**
Es posa a la classe base, a continuació de l'anotació **@Inheritance**. Indica les característiques del camp que indicarà la classe de l'objecte. Els valors que poden indicar-se són:
 - *nomColumna* és el nom que es dona al camp de la base de dades. La dada és opcional i el seu valor per defecte és **"DTYPE"**.
 - *tipus* és el tipus del camp. És opcional. El seu valor per defecte és **STRING**. A més, pot tenir els valors **CHAR** i **INTEGER**.
 - *mida*: també és opcional; només es té en compte si el tipus és **STRING** i indica la mida màxima de les cadenes de caràcter. Si s'indica amb un altre tipus, el valor és ignorat.
- **@DiscriminatorValue(valor)** Es pot indicar tant a la classe base com a les subclasses. És opcional. Es posa a continuació de les anotacions anteriors o, si no n'hi ha cap, a continuació de l'anotació **@Entity**. Indica el valor que prendrà el camp de la base de dades que indica la classe dels objectes per aquesta classe en concret. Si no es posa, pren un valor per defecte. Aquest valor per defecte, si la classe del camp és **STRING**, coincideix amb el nom de la classe. En cas contrari, depèn de la biblioteca.

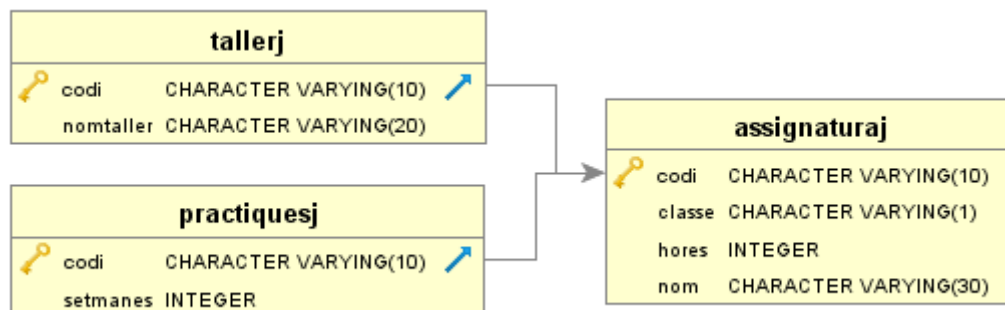
Exemple:



S'ha implementat amb les tres estratègies. Per distingir-les clarament, a cada exemple s'ha afegit la inicial de l'estratègia a continuació del nom de cada classe. També s'han destacat amb fons vermell les anotacions relacionades amb l'herència.

Estratègia *Joined*

Taules generades:



Classe base:

```

@Entity
@Inheritance(strategy=JOINED)
@DiscriminatorColumn(name="classe",
    discriminatorType=STRING, length=1)
@DiscriminatorValue("A")

public class AssignaturaJ
    implements Serializable {

    private String codi;
    private String nom;
    private int hores;

    @Id
    @Column(length=10)
    public String getCodi() {
        return codi;
    }

    public void setCodi(String codi) {
        this.codi = codi;
    }

    @Column(length=30)
    public String getNom() {
        return nom;
    }
}
  
```

Subclasses:

```

@Entity
@DiscriminatorValue("P")
public class PractiquesJ
    extends AssignaturaJ{

    private int setmanes;

    public int getSetmanes() {
        return setmanes;
    }

    public void setSetmanes
        (int setmanes) {
        this.setmanes = setmanes;
    }
}
  
```

<pre> public void setNom(String nom) { this.nom = nom; } public int getHores() { return hores; } public void setHores(int hores) { this.hores = hores; } </pre>	<pre> @Entity @DiscriminatorValue("T") public class TallerJ extends AssignaturaJ{ private String nomTaller; @Column(length=20) public String getNomTaller() { return nomTaller; } public void setNomTaller(String nomTaller) { this.nomTaller = nomTaller; } } </pre>
--	--

Estratègia *SINGLE_TABLE*

```

@Entity
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="classe",
discriminatorType=STRING, length=1)
@DiscriminatorValue("A")
public class AssignaturaS
    implements Serializable {

    private String codi;
    private String nom;
    private int hores;


    @Id
    @Column(length=10)
    public String getCodi() {
        return codi;
    }

    public void setCodi(String codi) {
        this.codi = codi;
    }

    @Column(length=30)
    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }
}

```

assignaturas		
	codi	CHARACTER VARYING(10)
	classe	CHARACTER VARYING(1)
	hores	INTEGER
	nom	CHARACTER VARYING(30)
	setmanes	INTEGER
	nomtaller	CHARACTER VARYING(20)

```

@Entity
@DiscriminatorValue("P")
public class PractiquesS
    extends AssignaturaS{










    private int setmanes;
    public int getSetmanes() {
        return setmanes;
    }

    public void setSetmanes(int setmanes) {
        this.setmanes = setmanes;
    }
}

```

<pre> public int getHores() { return hores; } public void setHores(int hores) { this.hores = hores; } </pre>	<pre> @Entity @DiscriminatorValue("T") public class TallerS extends PractiquesS{ private String nomTaller; @Column(length=20) public String getNomTaller() { return nomTaller; } public void setNomTaller (String nomTaller) { this.nomTaller = nomTaller; } } </pre>
--	---

Estratègia *TABLE_PER_CLASS*

<table border="1"> <thead> <tr> <th colspan="2">assignaturat</th> </tr> </thead> <tbody> <tr> <td></td> <td>codi CHARACTER VARYING(10)</td> </tr> <tr> <td></td> <td>hores INTEGER</td> </tr> <tr> <td></td> <td>nom CHARACTER VARYING(30)</td> </tr> </tbody> </table>	assignaturat			codi CHARACTER VARYING(10)		hores INTEGER		nom CHARACTER VARYING(30)	<table border="1"> <thead> <tr> <th colspan="2">practiquet</th> </tr> </thead> <tbody> <tr> <td></td> <td>codi CHARACTER VARYING(10)</td> </tr> <tr> <td></td> <td>hores INTEGER</td> </tr> <tr> <td></td> <td>nom CHARACTER VARYING(30)</td> </tr> <tr> <td></td> <td>setmanes INTEGER</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th colspan="2">tallert</th> </tr> </thead> <tbody> <tr> <td></td> <td>codi CHARACTER VARYING(10)</td> </tr> <tr> <td></td> <td>hores INTEGER</td> </tr> <tr> <td></td> <td>nom CHARACTER VARYING(30)</td> </tr> <tr> <td></td> <td>nomtaller CHARACTER VARYING(20)</td> </tr> <tr> <td></td> <td>setmanes INTEGER</td> </tr> </tbody> </table>	practiquet			codi CHARACTER VARYING(10)		hores INTEGER		nom CHARACTER VARYING(30)		setmanes INTEGER	tallert			codi CHARACTER VARYING(10)		hores INTEGER		nom CHARACTER VARYING(30)		nomtaller CHARACTER VARYING(20)		setmanes INTEGER
assignaturat																															
	codi CHARACTER VARYING(10)																														
	hores INTEGER																														
	nom CHARACTER VARYING(30)																														
practiquet																															
	codi CHARACTER VARYING(10)																														
	hores INTEGER																														
	nom CHARACTER VARYING(30)																														
	setmanes INTEGER																														
tallert																															
	codi CHARACTER VARYING(10)																														
	hores INTEGER																														
	nom CHARACTER VARYING(30)																														
	nomtaller CHARACTER VARYING(20)																														
	setmanes INTEGER																														
<pre> @Entity @Inheritance(strategy=TABLE_PER_CLASS) public class AssignaturaT implements Serializable { private String codi; private String nom; private int hores; @Id @Column(length=10) public String getCodi() { return codi; } public void setCodi(String codi) { this.codi = codi; } } </pre>	<pre> @Entity public class PractiquesT extends AssignaturaT{ private int setmanes; public int getSetmanes() { return setmanes; } public void setSetmanes(int setmanes) { this.setmanes = setmanes; } } </pre>																														

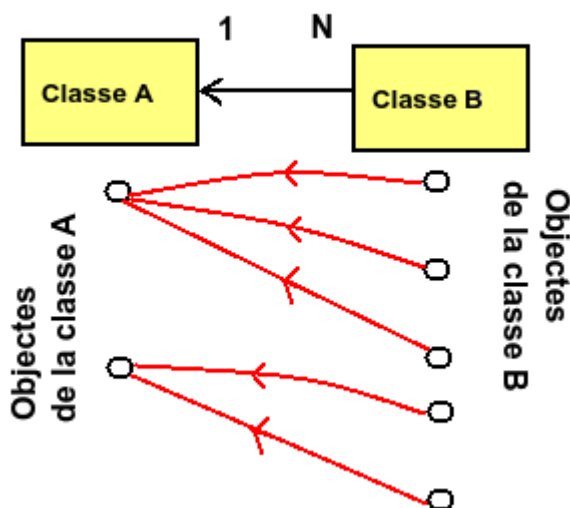
<pre> @Column(length=30) public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } public int getHores() { return hores; } public void setHores(int hores) { this.hores = hores; } </pre>	<pre> @Entity public class TallerT extends PractiquesT{ private String nomTaller; @Column(length=20) public String getNomTaller() { return nomTaller; } public void setNomTaller (String nomTaller) { this.nomTaller = nomTaller; } } </pre>
--	---

6. Definint relacions entre classes persistents i les seves claus amb anotacions

a) Tipus de relacions:

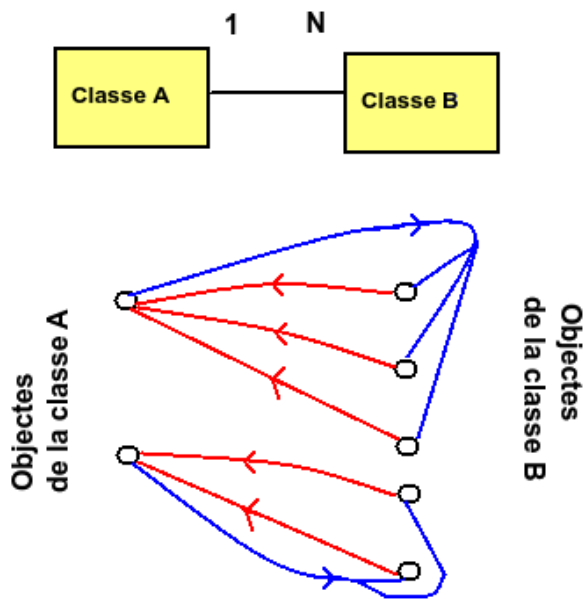
Les relacions entre les entitats, igual que passa en el model de bases de dades CHEN, poden tenir les cardinalitats *un-a-un*, *un-a-molts*, *molts-a-un* o *molts-a-molts*. L'altra característica important és la *navegabilitat*. La *navegabilitat* és la capacitat que tenen els objectes d'una classe per accedir als objectes a la classe amb la qual estan relacionats. Aquesta pot ser en un sentit o en tots dos sentits. En el primer cas, la relació es representa com una fletxa que surt de la classe els objectes de la qual tenen accés als objectes relacionats. En el segon cas, la relació es representa com una línia sense puntes de fletxa a cap dels dos costats.

Gràficament:



Relació 1-N navegable en una direcció. Concretament, de la part N (classe B) a la part 1 (classe A).

Els objectes de la classe B tindran una referència a l'objecte de la classe A amb què estan relacionats.



Relació 1-N navegable en totes dues direccions.

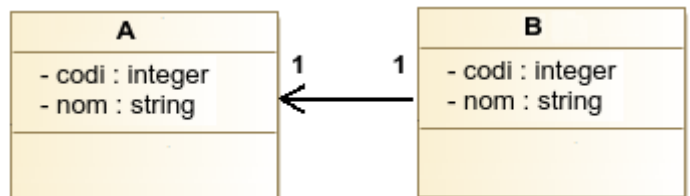
Els objectes de la classe B tindran, igual que abans, una referència a l'objecte de la classe A amb què estan relacionats.

Els objectes de la classe A tindran, una llista de referències, una a cadascun dels objectes amb què estan relacionats.

Segons la navegabilitat i la cardinalitat de la relació cal definir o no unes determinades dades i posar-hi unes anotacions o unes altres.

b) Relacions 1 a 1 navegables en un sentit

Cada objecte de la classe B tindrà una dada per accedir a l'objecte de la classe A amb el qual es relaciona. Aquesta dada no es representa explícitament al diagrama de classes



Els objectes de la classe A no tindran cap dada a part de les que apareixen al diagrama, doncs no poden accedir a l'objecte amb el qual es relacionen.

La classe A només tindrà les anotacions pròpies de les entitats (classes persistents). La classe B, a més, tindrà la dada necessària per accedir als objectes relacionats. A aquesta dada es vincularà l'anotació **@OneToOne** que, a més, indica la cardinalitat de la relació.

Aquesta dada i l'anotació **@OneToOne** s'han destacat posant-les en una línia amb fons groc.


```
@Entity
public class A
    implements Serializable{
    private int codi;
    private String nom;

    public A() {
    }

    public A(int codi, String nom) {
        this.codi = codi;
        this.nom = nom;
    }

    @Id
    public int getCodi() {
        return codi;
    }

    public void setCodi(int codi) {
        this.codi = codi;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

```
@Entity
public class B implements Serializable{
    private int codi;
    private String nom;
    private A relacionat;

    public B() {
    }

    public B(int codi, String nom, A relacionat)
    {
        this.codi = codi;
        this.nom = nom;
        this.relacionat=relacionat;
    }

    @Id
    public int getCodi() {
        return codi;
    }

    public void setCodi(int codi) {
        this.codi = codi;
    }

    public String getNom() {
        return nom;
    }

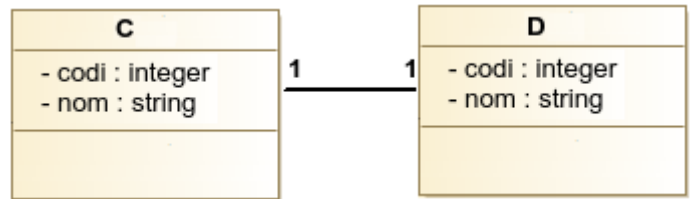
    public void setNom(String nom) {
        this.nom = nom;
    }

    @OneToOne
    public A getRelacionat() {
        return relacionat;
    }

    public void setRelacionat(A relacionat) {
        this.relacionat = relacionat;
    }
}
```

c) Relacions 1 a 1 navegables en tots dos sentits

Cada objecte de la classe C tindrà una dada per accedir a l'objecte de la classe D amb el qual es relaciona. Aquesta dada no es representa explícitament al diagrama de classes.



Igualment, cada objecte de la classe D tindrà una dada per accedir a l'objecte de la classe C amb el qual es relaciona. Aquesta dada tampoc es representa explícitament al diagrama de classes.

Totes dues classes tindran les anotacions pròpies de les entitats. A més, totes dues tindran una dada més i l'anotació **@OneToOne**.

Hi ha, però, una dificultat: Pot passar que l'objecte c1 de la classe C tingui una referència a l'objecte d1 de la classe D, però la referència d'aquest, en lloc de ser a c1 sigui a c2? Doncs,... sí que pot passar, però comportaria una inconsistència.

JPA afronta aquesta situació de la següent manera: a memòria totes dues classes tenen la dada que la relaciona amb l'objecte de l'altra classe. A la base de dades, però, només s'emmagatzema la informació de la relació d'un dels dos objectes. L'altra dada només existeix a memòria i s'omple automàticament en recuperar l'objecte de la base de dades.

Això es denota posant a la classe la dada de la qual no es grava l'anotació **@OneToOne** de la següent manera: **@OneToOne(mappedBy="dada")** on "camp" és el nom de la dada de l'altra classe que sí s'emmagatzema a la base de dades.

En aquest cas, totes dues classes tindran, a més de les anotacions pròpies de les entitats, la dada necessària per accedir als objectes relacionats de l'altra classe. Cadascuna d'aquestes dades anirà vinculada a una anotació **@OneToOne**. Al cas de la dada que no es gravarà, l'anotació portarà el paràmetre *mappedBy*.

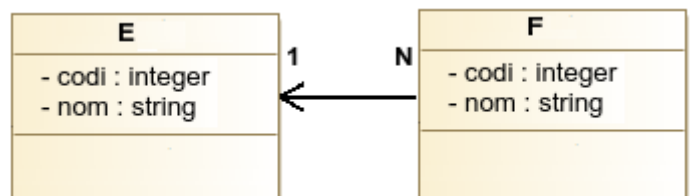
Les dades que relacionen objectes de les dues classes i l'anotació **@OneToOne** s'han destacat posant-les en una línia amb fons groc.

<pre> @Entity public class C implements Serializable{ private int codi; private String nom; private D relacionat; public C() { } public C(int codi, String nom, D relacionat) { this.codi = codi; this.nom = nom; this.relacionat=relacionat; } </pre>	<pre> Entity public class D implements Serializable{ private int codi; private String nom; private C vinculat; public D() { } public D(int codi, String nom, C relacionat) { this.codi = codi; this.nom = nom; this.vinculat=relacionat; } </pre>
--	---

<pre> @Id public int getCodi() { return codi; } public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @OneToOne public D getRelacionat() { return relacionat; } public void setRelacionat(D relacionat) { this.relacionat = relacionat; } </pre>	<pre> @Id public int getCodi() { return codi; } public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @OneToOne(mappedBy="relacionat") public C getVinculat() { return vinculat; } public void setVinculat (C vinculat) { this.vinculat = vinculat; } </pre>
---	---

d) Relacions 1 a N navegables d'N a 1

Cada objecte de la classe F tindrà una dada per accedir a l'objecte de la classe E amb el qual es relaciona. Aquesta dada no es representa explícitament al diagrama de classes



Els objectes de la classe E no tindran cap dada a part de les que apareixen al diagrama, doncs no poden accedir als objectes amb els quals es relacionen.

La classe E només tindrà les anotacions pròpies de les entitats (classes persistents). La classe F, a més, tindrà la dada necessària per accedir als objectes relacionats. A aquesta dada es vincularà l'anotació **@ManyToOne** que, a més, indica la cardinalitat de la relació.

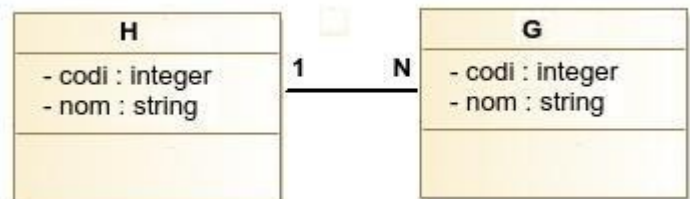
Aquesta dada i l'anotació **@ManyToOne** s'han destacat posant-les en una línia amb fons groc.

<pre> @Entity public class E implements Serializable{ private int codi; private String nom; public E() { } public E(int codi, String nom) { this.codi = codi; this.nom = nom; } @Id public int getCodi() { return codi; } public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } } </pre>	<pre> @Entity public class F implements Serializable{ private int codi; private String nom; private E relacionat; public F() { } public F(int codi, String nom, E relacionat) { this.codi = codi; this.nom = nom; this.relacionat=relacionat; } @Id public int getCodi() { return codi; } public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @ManyToOne public E getRelacionat() { return relacionat; } } </pre>
--	--

```
public void setRelacionat (E relacionat) {
    this.relacionat = relacionat;
}
}
```

e) Relacions 1 a N navegables en tots dos sentits

Cada objecte de la classe G tindrà una dada per accedir a l'objecte de la classe H amb el qual es relaciona. Aquesta dada no es representa explícitament al diagrama de classes.



Igualment, cada objecte de la classe H tindrà una llista per accedir als objectes de la classe G amb el qual es relaciona. Aquesta llista tampoc es representa explícitament al diagrama de classes.

Totes dues classes tindran les anotacions pròpies de les entitats. A més, la classe H tindrà la llista esmentada abans vinculada a l'anotació **@OneToMany** i la classe G tindrà una dada per accedir a l'objecte de la classe H amb el qual està relacionat i aquesta dada anirà vinculada amb l'anotació **@ManyToOne**.

Igual que passava amb la relació 1 a 1 navegable en tots dos sentits, hi ha un problema de redundància de dades: els objectes de cada classe tenen la seva pròpia informació sobre la relació i aquesta pot ser contradictòria.

JPA evita aquest problema novament escrivint en la classe H l'anotació **@OneToMany** de la següent manera: **@OneToMany(mappedBy="dada")**; "dada" és la dada dels objectes de la classe G que conté l'objecte relacionat de la classe H.

El funcionament és el següent: la informació de les relacions només s'actualitza quan es grava un objecte de la classe G (no té la part *mappedBy*). Quan es grava un objecte de la classe H, aquesta informació no es modifica. Ara bé, quan es recupera un objecte de la classe H, la seva llista amb els objectes relacionats s'omple amb la informació de la relació que hi ha a la base de dades.

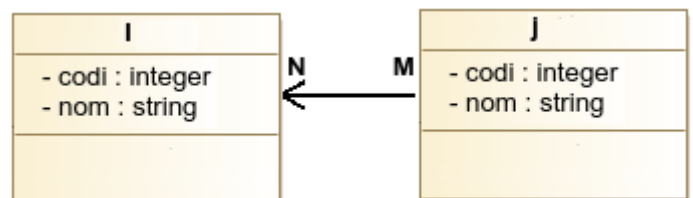
Les dades que no apareixen al diagrama de classes i les anotacions **@OneToMany** i **@ManyToOne** s'han destacat posant-les en una línia amb fons groc.

@Entity public class G implements Serializable{ private int codi; private String nom; private H vinculat; public G() { }	@Entity public class H implements Serializable{ private int codi; private String nom; private List<G> relacionats= new ArrayList<>(); public H() { }
--	--

<pre> public G(int codi, String nom,H vinculat){ this.codi = codi; this.nom = nom; this.vinculat=vinculat; } @Id public int getCodi() { return codi; } public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @ManyToOne public H getVinculat() { return vinculat; } public void setVinculat(H vinculat) { this.vinculat = vinculat; } </pre>	<pre> public H(int codi, String nom) { this.codi = codi; this.nom = nom; } @Id public int getCodi() { return codi; } public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @OneToMany(mappedBy="vinculat") public List<G> getRelacionats() { return relacionats; } public void setRelacionats(List<G> relacionats){ this.relacionats = relacionats; } </pre>
--	---

f) Relacions M a N navegables en un sol sentit

Cada objecte de la classe J tindrà una llista per accedir als objectes de la classe I amb els quals es relaciona. Aquesta dada no es representa explícitament al diagrama de classes



Els objectes de la classe I no tindran cap dada a part de les que apareixen al diagrama, doncs no poden accedir als objectes amb els quals es relacionen.

La classe I només tindrà les anotacions pròpies de les entitats (classes persistents). La classe J, a més, tindrà la llista necessària per accedir als objectes relacionats. A aquesta llista es vincularà l'anotació **@ManyToMany** que, a més, indica la cardinalitat de la relació.

Aquesta llista i l'anotació **@ManyToMany** s'han destacat posant-les en una línia amb fons groc.

```
@Entity
public class I implements Serializable{
    private int codi;
    private String nom;

    public I() {
    }

    public I(int codi, String nom) {
        this.codi = codi;
        this.nom = nom;
    }

    @Id
    public int getCodi() {
        return codi;
    }

    public void setCodi(int codi) {
        this.codi = codi;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

```
@Entity
public class J implements Serializable{

    private int codi;
    private String nom;
    private List<I> relacionats=
        new ArrayList<I>();

    public J() {
    }

    public J(int codi, String nom){
        this.codi = codi;
        this.nom = nom;
    }

    @Id
    public int getCodi() {
        return codi;
    }

    public void setCodi(int codi) {
        this.codi = codi;
    }

    public String getNom() {
        return nom;
    }

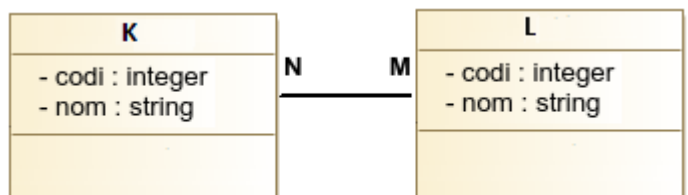
    public void setNom(String nom) {
        this.nom = nom;
    }

    @ManyToMany
    public List<I> getRelacionats() {
        return relacionats;
    }

    public void setRelacionats(
        List<I> relacionats) {
        this.relacionats = relacionats;
    }
}
```

g) Relacions M a N navegables en tots dos sentits

Cada objecte de la classe L tindrà una llista per accedir als objectes de la classe



K amb els quals es relaciona. Aquesta llista no es representa explícitament al diagrama de classes.

Igualment, cada objecte de la classe K tindrà una llista per accedir als objectes de la classe L amb el qual es relaciona. Aquesta llista tampoc es representa explícitament al diagrama de classes.

Totes dues classes tindran les anotacions pròpies de les entitats. A més, totes dues tindran també l'anotació **@ManyToMany**.

Com a totes les relacions navegables en tots dos sentits, apareix el problema de la redundància de les dades de la relació: els objectes de cada classe tenen la seva pròpia llista amb informació sobre la relació i aquesta pot ser contradictòria.

JPA evita aquest problema escrivint novament l'anotació **@ManyToMany** en una de les dues classes de la següent manera: **@ManyToMany(mappedBy="dada")**; "dada" és la dada de l'altra classe que conté la llista dels objectes relacionats.

El funcionament és el següent: la informació de les relacions només s'actualitza quan es grava un objecte de la classe que té l'anotació **@ManyToMany** (sense *mappedBy*), en el nostre cas, serà la classe K. Quan es grava un objecte de l'altra classe (L a l'exemple), aquesta informació no es modifica a la base de dades. Ara bé, quan es recupera un objecte de la classe L, la seva llista amb els objectes relacionats s'omple amb la informació de la relació que hi ha a la base de dades.

Les dades que no apareixen al diagrama de classes i les anotacions **@ManyToMany** s'han destacat posant-les en una línia amb fons groc.

<pre>@Entity public class K implements Serializable{ private int codi; private String nom; private List<L> vinculats= new ArrayList<>(); public K() { } public K(int codi, String nom) { this.codi = codi; this.nom = nom; } @Id public int getCodi() { return codi; } }</pre>	<pre>@Entity public class L { private int codi; private String nom; private List<K> relacionats= new ArrayList<>(); public L() { } public L(int codi, String nom) { this.codi = codi; this.nom = nom; } @Id public int getCodi() { return codi; } }</pre>
---	--

<pre> public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @ManyToMany public List<L> getVinculats() { return vinculats; } public void setVinculats (List<L> vinculats) { this.vinculats = vinculats; } </pre>	<pre> public void setCodi(int codi) { this.codi = codi; } public String getNom() { return nom; } public void setNom(String nom) { this.nom = nom; } @ManyToMany(mappedBy="vinculats") public List<K> getRelacionats() { return relacionats; } public void setRelacionats (List<K> relacionats) { this.relacionats = relacionats; } </pre>
--	--

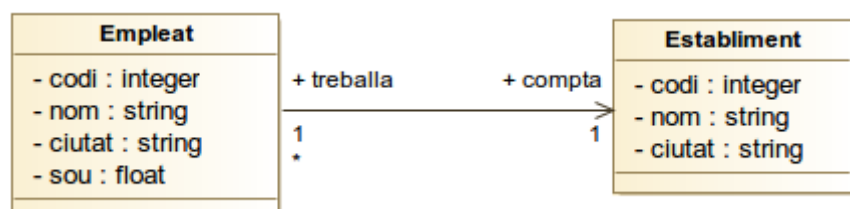
7. JPQL. Definició de consultes amb nom amb anotacions. JPQL encastat.

a) Introducció a JPQL

JPQL és el DML de JPA. Té una sintaxi similar a SQL, però treballa amb objectes en lloc de fer-ho amb taules. Això, per exemple, estalvia haver de fer *joins*.

Tenim aquestes classes:

La dada d'empleat que permet accedir des d'un empleat al seu establiment, s'anomena *establiment* i es declara de la classe Establiment.



Exemples de consultes:

SELECT e FROM Empleat e	Obté tots els empleats
SELECT e FROM Empleat e WHERE e.nom="Lluïsa"	Obté totes les empleades anomenades Lluïsa
SELECT e FROM Empleat e WHERE e.establiment.codi=1	Obté tots els empleats de l'establiment identificat amb el codi 1

Formació professional

SELECT e FROM Empleat e WHERE e.ciutat=e.establiment.ciutat	Obté tots els empleats que treballen en un establiment de la seva mateixa ciutat.
SELECT e.nom, e.ciutat FROM Empleat e WHERE e.codi=2	Obté el nom i la ciutat de residència de l'empleat amb codi 2.

Actualitzacions:

UPDATE Empleat e SET e.nom="Julian Perez" WHERE e.codi=10	Modifica el nom de l'empleat amb codi 10
UPDATE Empleat e SET e.sou=e.sou*1,10 WHERE e.ciutat="Badalona"	Incrementa en un 10% el sou dels empleats que viuen a Badalona
UPDATE Empleat e SET e.sou=e.sou*1,10 WHERE e.establiment.ciutat="Badalona"	Incrementa en un 10% el sou dels empleats que treballen a Badalona
UPDATE Establiment e SET e.codi=e.codi*10,e.ciutat="Montgat" WHERE e.ciutat="Tiana" and mod(e.codi,2)=1	Traslada tots els establiments amb codi senar de Tiana a Montgat i els recodifica multiplicant el seu codi per 10

Esborrat:

DELETE FROM Establiment e WHERE e.codi > 1000	Esborra tots els establiments amb un codi superior a 1000
---	---

Una diferència rellevant entre SQL i JPQL és que JPQL no té la instrucció INSERT. Els objectes s'afegeixen amb els mètodes *persist* i *merge* de la classe *EntityManager*.

Amb JPQL també poden utilitzar-se paràmetres, que seran substituïts per valors quan es faci la crida. Alguns exemples, basats en els anteriors, són:

SELECT e FROM Empleat e WHERE e.nom=:nom	Obté tots els empleats que es diuen igual que el paràmetre <i>nom</i>
UPDATE Empleat e SET e.nom=:nom WHERE e.codi=:codi	Assigna el nom indicat pel paràmetre <i>nom</i> a l'empleat amb codi <i>codi</i>
DELETE FROM Establiment e WHERE e.codi > :maxim	Esborra tots els establiments amb un codi superior al valor del paràmetre <i>maxim</i>

b) Definició de consultes amb nom

A JPA poden definir-se consultes amb nom, que després seran utilitzades quan s'utilitzi JPQL encastat. Es fa amb les anotacions **@NamedQueries** i **@NamedQuery**, que cal posar entre l'anotació **@Entity** i la declaració de la classe. Concretament:

```
@Entity
@NamedQueries({
    @NamedQuery(name="nom1", query="query1-en-JPQL"),
    @NamedQuery(name="nom2", query="query2-en-JPQL"),
    @NamedQuery(name="nom3", query="query3-en-JPQL"),
})
```

Observacions:

- L'última anotació `@NamedQuery` no porta coma després del parèntesi de tancar.
- El nom de cada *named query* sol ser *NomDeLaClasse.nomDelQuery*.
- Les queries poden portar paràmetres.

Exemple:

```
@Entity
@NamedQueries({
    @NamedQuery(name="Empleat.tots",query= "SELECT e FROM Empleat e"),
    @NamedQuery(name="Empleat.perNom",
        query= "SELECT e FROM Empleat e WHERE e.nom=:nom"),
    @NamedQuery(name="Empleat.perCiutat",
        query= "SELECT e FROM Empleat e WHERE e.ciutat=:ciutat"),
    @NamedQuery(name="Empleat.perEstabliment",
        query= "SELECT e FROM Empleat e WHERE e.establiment.codi=:codi"),
    @NamedQuery(name="Empleat.viuenOnTreballen",
        query= "SELECT e FROM Empleat e WHERE e.ciutat=e.establiment.ciutat")
})

public class Empleat implements Serializable {
    ....
}
```

c) JPQL encastat

Suposem que em és un *EntityManager* obert:

- Consultes (Select):

- Primer cal cridar a *em.createQuery* o *em.createNamedQuery* passant-li la consulta que volem realitzar. El resultat és un objecte de la interfície *Query*.
Per exemple:

```
Query q=em.createQuery("SELECT e FROM Establiment e WHERE e.codi=1");
Query nq=em.createNamedQuery("Empleat.tots"); // Empleat.tots s'ha definit a
// l'exemple anterior
```

- A continuació, ja es pot realitzar la consulta cridant a *getSingleResult*, si esperem només un resultat o a *getResultList*, si esperem una llista de resultats. Al primer cas, si el nombre de resultats de la consulta no és exactament 1, es produeix una **excepció**. Exemples (continuació dels anteriors):

```
Establiment es = (Establiment) q.getSingleResult(); // només hi ha un establiment
// amb codi igual a 1
List<Empleat> totsEmpleats = nq.getResultList();
```

- Modificacions (Update) i esborrats (Delete):

- Primer cal crear una query igual que abans i, a més, **iniciar una transacció**.

Per exemple:

```
// la següent Query pujarà el sou un 10% a tots els empleats  
Query qU=em.createQuery("UPDATE Empleat e SET e.sou=e.sou*1,10");  
  
// la següent Query esborrarà l'establiment amb codi = 1  
Query qE="DELETE FROM Establiment e WHERE e.codi=1";  
  
em.getTransaction().begin(); // es comença la transacció
```

- A continuació, cal executar el mètode `executeUpdate` i tancar la transacció. El mètode `executeUpdate` retorna el nombre d'objectes modificats o esborrats.

Continuant els exemples anteriors:

```
qU.executeUpdate(); // puja efectivament el sou als empleats el 10%  
                                     // (a l'espera del commit).  
qE.executeUpdate(); // esborra efectivament l'establiment amb codi = 1  
                                     //(a l'espera del commit).  
em.getTransaction().commit(); // confirma la transacció (fa el commit).
```

- Treball amb paràmetres:

Si la consulta té paràmetres, abans de cridar als mètodes `getSingleResult`, `getResultList` o `executeUpdate`, cal donar valor a aquests paràmetres. Es fa amb el mètode de `Query` `setParameter`.

Per exemple:

```
// la següent transacció incrementa el sou en quantitat (1er. paràmetre) euros a  
                                     // l'empleat  
// identificat pel número codi (2on. Paràmetre)  
  
Query q=em.createQuery("UPDATE Empleat e SET e.sou=e.sou+:quantitat WHERE  
e.codi=:codi");  
  
// Donem valor als paràmetres:  
// en aquest cas, s'incrementa el sou en 80 euros al treballador amb codi 7  
  
q.setParameter("quantitat",80);  
q.setParameter("codi",7);  
em.getTransaction().begin(); // abans de fer una modificació, cal haver començat  
                                     // una transacció  
  
q.executeQuery(); // es realitza l'actualització  
em.getTransaction().commit(); // es confirma la transacció
```

8. Especificació de la persistència sense anotacions

Utilitzar anotacions directament sobre les classes per especificar la seva persistència és còmode, però també té un inconvenient important: no sempre volem o podem posar anotacions sobre les classes. Per exemple, pot ser que treballem amb unes classes que no es poden modificar perquè es comparteixen amb una altra aplicació, que, inclús, pot ser anterior a la nostra. També pot ser que no vulguem posar les anotacions a les classes per tenir més separada la lògica de l'aplicació de la seva persistència.

Sortosament, JPA permet també especificar la mateixa informació en un fitxer de configuració que té format XML.

Per fer-ho, cal seguir els següents passos:

1. Afegir al fitxer *persistence.xml* l'element **<mapping-file>***nomDelFitxer***</mapping-file>** on *nomDelFitxer* és el nom i ubicació del fitxer on anirà l'especificació de la persistència de les classes que ho han de ser (les *Entity*). Aquest element ha d'anar a l'inici de la *persistence-unit*.

Per exemple, si volem especificar que l'especificació corresponent a la unitat de persistència *Unitat1* estarà al fitxer *orm1.xml* i que aquest es trobarà a la mateixa carpeta que *persistence.xml* es posaria així:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">

  <persistence-unit transaction-type="RESOURCE_LOCAL" name="Unitat1">

    <mapping-file>META-INF/orm1.xml</mapping-file>

    .....
  </persistence-unit>
  .....
</persistence>
```

2. Complimentar el fitxer amb l'especificació. Al nostre exemple, el fitxer META-INF/orm1.xml. Té la següent estructura:

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings version="2.0"
xmlns:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/persistence/orm">

.....
</entity-mappings>
```

On hi ha els punts suspensius cal anar reproduint les diferents anotacions traduïdes al format XML que assenyalen l'especificació.

És opcional fer-ho, però, podem anul·lar qualsevol possible especificació amb anotacions que hi hagi a les classes si **just a continuació** de l'element **<entity-mappings...>** posem els elements següents:

```
<persistence-unit-metadata>
  <xml-mapping-metadata-complete/>
</persistence-unit-metadata>
```

a) Una entitat

L'equivalent a

```
package model;

@Entity
public class Assignatura implements Serializable{
.....
}
```

És això:

```
<entity class="model.Assignatura" metadata-complete="valor">
.....
</entity>
```

Cal tenir present que el nom de la classe s'ha de posar precedit pel paquet al qual pertany.

On hi ha els punts suspensius hi va la informació sobre la classe en aquest ordre:

- especificació de l'herència
- especificació de les consultes amb nom (*named queries*)
- especificació dels camps (clau inclosa) i les relacions

Tots els punts són opcionals. Poden ometre's perquè no calgui especificar res. Si és adient especificar alguna cosa i no es posen, es tindrà en compte la implementació per defecte o l'especificada amb anotacions a la classe, tenint, a més, en compte el valor de l'atribut **metadata-complete**.

L'atribut **metadata-complete** pot prendre els valors **"true"** o **"false"**. El valor **"true"** indica que sobreescrivim totalment qualsevol especificació que contingui la classe mitjançant anotacions. Si el valor és **"false"** o no posem l'atribut **metadata-complete**, es combinen totes dues especificacions (les realitzades amb anotacions i les realitzades amb XML). En cas de contradicció, tenen prioritat les especificacions realitzades amb XML.

Si vulguéssim especificar el nom de la taula, caldria afegir, a continuació de l'element *entity* l'equivalent a l'anotació **@Table(name="nomDeLaTaula")** de la següent manera:

```
<entity ...
  <table name="nomDeLaTaula"/>
.....
</entity>
```

b) La implementació de l'herència

Anotació	@Inheritance(strategy=<i>estratègia</i>)
Element XML equivalent	<inheritance strategy="estratègia"/>

Les estratègies s'escriuen així: "SINGLE TABLE", "TABLE PER CLASS" o "JOINED"

Anotació	@DiscriminatorColumn(name="<i>nomColumna</i>", discriminatorType=<i>tipus</i>, length=<i>mida</i>)
Element XML equivalent	<discriminator-column name="<i>nomColumna</i>" discriminator-type="<i>tipus</i>", length="<i>mida</i>" />

Els tipus del discriminador s'escriuen així: "STRING", "CHAR" o "INTEGER"

Anotació	@DiscriminatorValue(<i>valor</i>)
Element XML equivalent	<discriminator-value> <i>valor</i> </discriminator-value>

c) Consultes amb nom (*named queries*)

Anotació	@NamedQueries({...})
Element XML equivalent	

No cal posar-hi res

Anotació	@NamedQuery(name="<i>nomConsulta</i>", query="<i>consultaEnJPQL</i>")
Element XML equivalent	<named-query name="<i>nomConsulta</i>"> <query> <i>consultaEnJPQL</i> </query> </named-query>

d) Camps i relacions

Camps

Totes les anotacions referides als camps i les relacions entre aquests van dins de l'element **<attributes>** **</attributes>**

"*nomDada*" fa referència a la dada declarada a la classe.

Anotació	@Id
Element XML equivalent	<id name=" <i>nomDada</i> " />

Anotació	@Column(name=" <i>nomColumna</i> ", length= <i>mida</i>)
Element XML equivalent	<basic name=" <i>nomCamp</i> " /> <column name=" <i>nomColumna</i> " length=" <i>mida</i> " /> </basic>

A l'element *column* només cal posar els atributs que ens interressi definir. Si no volem definir-ne cap, no cal posar l'element XML.

Relacions

A tots els requadres, "*nomDada*" fa referència a la dada declarada a la classe que estem especificant, mentre que "*dada*" fa referència a la dada de la classe relacionada amb la primera

Anotació	@OneToOne
Element XML equivalent	<one-to-one name=" <i>nomDada</i> " />

Anotació	@OneToOne(mappedBy=" <i>dada</i> ")
Element XML equivalent	<one-to-one name=" <i>nomDada</i> " mappedBy=" <i>dada</i> " />

Anotació	@OneToMany
Element XML equivalent	<one-to-many name=" <i>nomDada</i> " />

Anotació	@OneToMany(mappedBy="dada")
Element XML equivalent	<one-to-many name="nomDada" mappedBy="dada"/>

Anotació	@ManyToOne
Element XML equivalent	<many-to-many name="nomCamp"/>

Anotació	@ManyToMany
Element XML equivalent	<many-to-many name="nomDada"/>

Anotació	@ManyToMany(mappedBy="dada")
Element XML equivalent	<many-to-many name="nomDada" mappedBy="dada"/>

Anotació	@ManyToMany @JoinTable(name="nomDeLaTaula")
Element XML equivalent	<many-to-many name="nomDada"/> <join-table name="nomDeLaTaula"/> </many-to-many>