Accés a dades.

Object Relational Mapping:

Java Persistence API

Object Relational Mapping

La gestió de dades en programació orientada a objectes es fa a través de la manipulació d'objectes, que quasi sempre són valors composts, un objecte sol tenir diversos atributs que poden ser altres objectes amb els seus atributs, ...

Les bases de dades relacionals només poden emmagatzemar valors escalars, com cadenes de caràcters o enters i organitzar-los en taules.

El programador ha de convertir els valors representats en forma d'objectes en valors simples o agrupats per l'emmagatzematge a la base de dades i implementar el procés invers. Alternativament podria treballar només amb valors escalars, perdent els avantatges de la Programació Orientada a Objectes.

Per poder gaudir de la potència de la programació a objectes i estalviar els processos de conversió d'objecte a registre de la base de dades i el seu invers, s'ha creat l'automatització d'aquest procés, mitjançant el mapatge d'objecte a relacional, la definició de les correspondències entre objectes i els seus atributs i taules i les seves columnes.

En teoria la missió d'un paquet ORM, una vegada configurat, és que el programador treballi amb els objectes del seu llenguatge i s'oblidi completament de la base de dades, no hauria de pensar en *selects* o *updates*, sinó en crear objectes i accedir als seus atributs. L'ORM seria el que s'adonàs que fa falta fer un *select* o un *insert*, el construiria, l'executaria i tornaria l'objecte demanat.

A la pràctica sempre hi ha una certa presència del paquet ORM, mai és completament transparent, i pot introduir penalitzacions en el rendiment de l'aplicació.

Els objectius reals de fer servir una eina ORM, són el de guanyar temps, simplificar el desenvolupament (és a dir, l'eina ORM absorbeix la part complexa que tenia el desenvolupador), incrementar el rendiment i l'escalabilitat i disminuir la complexitat en l'arquitectura.

La major crítica que reben aquests paquets és basa en l'origen del problema: traduir els objectes a un model relacional. Si hi ha bases de dades orientades a objectes que no necessiten aquesta traducció perquè no utilitzar-les? Les bases de dades orientades a objectes no defineixen taules, sinó tipus, fent el mapatge innecessari.

Java Persistence API

L'Api de persistència de Java proporciona als desenvolupadors Java una eina de mapatge objecte / relacional per a poder utilitzar bases de dades relacionals dins les aplicacions Java. Està formada per quatre components:

- L'API de persistència
- El Java Persistence Query Language, llenguatge de consultes similar a l'SQL, però utilitzant els objectes de l'aplicació en lloc de les taules de la base de dades.
- L'Api Criteria, una altra manera de fer consultes a la base de dades.
- Metadades per al mapatge objecte / relacional.

Tot seguit farem una introducció als conceptes i les eines necessaris per utilitzar-la a les nostres aplicacions.

Unitat de persistència

Una unitat de persistència és una col·lecció de propietats amb un nom únic que utilitzarem a la nostra aplicació per determinar com gestionarem unes determinades entitats (les classes que representen la base de dades) dins la nostra aplicació i com les guardarem a la base de dades.

Si volem utilitzar JPA a la nostra aplicació necessitam al manco una unitat de persistència. En podem definir més d'una, però totes han de tenir un nom únic.

Les unitats de persistència es defineixen dins el fitxer *persistence.xml*. En una aplicació JSE es sol situar al directori *META-INF*.

Algunes de les propietats que es guarden a una unitat de persistència són:

- Les classes d'entitat gestionades per la unitat de persistència.
- El proveïdor de persistència, la llibreria utilitzada per realitzar el pas d'entitats a taules de la base de dades relacional. Per exemple Hibernate. Al projecte haurem d'incloure les llibreries necessàries per utilitzar-lo.
- Les dades de la connexió de base de dades: url, usuari, password, classe del driver, ... o del DataSource del servidor.
- El tipus de transacció utilitzat: JTA en aplicacions sobre un servidor JEE o RESOURCE_LOCAL en una aplicació JSE.

Entity classes

Ja hem dit que si utilitzam JPA tendrem objectes a les nostres aplicacions que representaran les tuples de la base de dades. Les classes que defineixen aquest objectes es diuen *Entity classes*. Abans aquestes classes eren un tipus més d'EJB. Fa unes quantes versions de Java, però, que es varen treure de l'especificació JEE per poder-se utilitzar també en aplicacions JSE.

Una classe d'entitat és un POJO amb un constructor sense paràmetres que representa una taula de la base de dades dins l'aplicació.

Aquesta classe tendrà un atribut per a cada camp de la base de dades que volguem utilitzar.

Les entitats han d'implementar els mètode *equals* i *hashCode*.

Anotacions

El mapatge, tant a la classe com als atributs, es fa amb anotacions. Algunes de les anotacions més habituals:

- @Entity Marca la classe com a entitat. Es posa a sobre de la definició de classe.
- @Table(name = "AUTORS", schema="biblioteca", catalog="") Defineix a quina taula es mapa la classe. Es posa seguint l'anotació Entity
- @Column(name = "NOM_AUT") Defineix a quin camp de la taula s'associa l'atribut declarat just després. Pot incloure altres atributs, com *nullable* o *length*.
- @Id defineix l'atribut com a clau primària de la taula.

Claus foranes. Relacions.

Per indicar una relació hem d'anotar els atributs de les dues classes que intervenen a la relació.

Relació 1-1

Per exemple la taula *Usuari* té una clau forana relacionada amb la taula *Adreça*.

- Al costat de la relació que representa la taula que té el camp amb la foreign key:
    - @OneToOne Defineix la relació com a un a un, és a dir.
    - @JoinColumn(name = "FK_ADREÇA", referencedColumnName = "IDADREÇA") Defineix la columna a que fa referència a l'altra taula.
- Si la relació és bidireccional, a la classe que representa la taula de l'altra costat de la relació, en aquest cas *Adreça*, tendrem un atribut *Usuari*. *MappedBy* indica l'atribut de la classe referenciada.
    - @OneToOne(mappedBy = "adreca")

```java
@Entity
@Table(name = "Usuaris", schema = "proves", catalog = "")
public class Usuari {
    private int idUsuari;
    private String nom;
    private Adreca adreca;

    @OneToOne
    @JoinColumn(name = "fk_adreca", referencedColumnName = "idAdreca")
    public Adreca getAdreca() { return adreca; }
```

Usuari › getAdreca()

```java
@Entity
@Table(name = "Adreces", schema = "proves", catalog = "")
public class Adreca {
    private int idAdreca;
    private String carrer;
    private String codiPostal;
    private String poblacio;
    private Usuari usuari;


    @OneToOne(mappedBy = "adreca")
    public Usuari getUsuari() { return usuari; }
```

Relació 1-n

- Al costat *n* d'una relació *1 a n*. La classe tendrà un atribut del tipus de la classe de l'altre costat de la relació anotat amb:
    - @ManyToOne Defineix la relació com a molts a un, és a dir, aquest atribut representa la part n d'una relació 1 a n.
    - @JoinColumn(name = "FK_NACIONALITAT", referencedColumnName = "NACIONALITAT") Defineix la columna a que fa referència a l'altra taula.
- Al costat *1* d'una relació *1 a n* es pot fer una llista que contengui tots els objectes associats. *MappedBy* indica l'atribut de la classe referenciada.
    - @OneToMany(mappedBy = "nacionalitat")

```java
@Entity
@Table(name = "AUTORS", schema = "biblioteca")
public class Autor {
    private int idAut;
    private String nomAut;
    private Timestamp dNaixAut;
    private Nacionalitat nacionalitat;

    @ManyToOne
    @JoinColumn(name = "FK_NACIONALITAT",
            referencedColumnName = "NACIONALITAT")
    public Nacionalitat getNacionalitat() { return nacionalitat; }
```

Autor › getNacionalitat()

nalitat.java

```java
@Entity
@Table(name = "NACIONALITATS", schema = "biblioteca")
public class Nacionalitat {
    private String nacionalitat;
    private Collection<Autor> autors;

    @OneToMany(mappedBy = "nacionalitat")
    public Collection<Autor> getAutors() { return autors; }
```

Relació n – m

Si aquestes relacions no contenen informació addicional (o no ens interessa utilitzar-la) podem mapejar la relació directament, de manera que no s'hagin de crear aquestes classes intermèdies. Per fer-ho:

- Cada taula de la relació ha de tenir un atribut que representi els objectes de l'altra costat de la relació.
- Per evitar duplicitats a la llista i a l'hora d'accedir a la base de dades, els atributs que representen aquestes relacions haurien de ser del tipus *Set*.
- A un costat de la relació, per exemple a la classe *Tema,* anotam l'atribut que representa l'altra costat de la relació, *llibres,* amb

  @ManyToMany(mappedBy = "nom de l'atribut a l'altra classe")

  Per exemple, a la classe *Tema*:

  ```
  @ManyToMany(mappedBy = "temes")
  private Set<Llibre> llibres;
  ```

- A l'altra costat de la relació, per exemple a la classe *Llibre* la cosa és un poc més complexe:
  - Anotam l'atribut *temes* amb @ManyToMany
  - Afegim l'anotació @JoinTable amb els següents atributs:
    - name: El nom de la taula que implementa la relació n-m, per exemple LLI_TEMA
    - joinColumns: El nom de les columnes que enllacen la taula amb la taula n-m. Per a cada columna tenim una anotació @JoinColumn amb dos atributs:
      - name: la columna de la taula n-m. Per exemple FK_IDLLIB
      - referencedName: La columna de la taula representada per la classe on hi ha la relació. Per exemple ID_LLIB

- inverseJoinColumns: Les columnes que representen l'altra costat de la relació. Per a cada columna tenim una anotació @JoinColumn amb dos atributs:
  - name: la columna de la taula n-m. Per exemple FK_TEMA
  - referencedName: La columna de la taula representada per la classe on hi ha la relació. Per exemple TEMA

Per exemple, a la classe *Llibre*

```
@ManyToMany
@JoinTable(name = "LLI_TEMA",
joinColumns = {
@JoinColumn(name = "FK_IDLLIB", referencedColumnName = "ID_LLIB")},
inverseJoinColumns = { @JoinColumn(name = "FK_TEMA",
referencedColumnName = "TEMA")})
private Collection<Tema> temaCollection;
```

ava × Autor.java × Llibre.java × persistence.xml ×

```java
@Entity
@Table(name = "LLIBRES", schema = "biblioteca", catalog = "")
public class Llibre {
    private int idLlib;
    private String titol;
    private String numEdicio;
    private Set<Tema> temes;

    @ManyToMany
    @JoinTable(name = "LLI_TEMA", catalog = "", schema = "biblioteca",
            joinColumns = @JoinColumn(name = "FK_IDLLIB",
                    referencedColumnName = "ID_LLIB", nullable = false),
            inverseJoinColumns = @JoinColumn(name = "FK_TEMA",
                    referencedColumnName = "TEMA", nullable = false))
    public Set<Tema> getTemes() { return temes; }
```

Llibre

java ×

```java
@Entity
@Table(name = "TEMES", schema = "biblioteca", catalog = "")
public class Tema {
    private String tema;
    private String temaPare;
    private Set<Llibre> llibres;

    @ManyToMany(mappedBy = "temes")
    public Set<Llibre> getLlibres() { return llibres; }
```

Fetch. Recuperació de les dades associades per una relació.

Podem definir el tipus de Fetch (eager o lazy), el moment en que es carreguen els objectes de l'altre costat de la relació. Ho podem posar a les anotacions @ManyToMany, @OneToMany, @ManyToOne, @OneToOne.

La diferència entre el fetch **eager** i **lazy** és el moment en el qual es carreguen a memòria des de la base de dades els elements d'un atribut, per exemple d'una col·lecció d'objectes associats per una relació *1 a n*.

Eager: Ansiòs. En carregar l'entitat amb l'objecte del costat *1* de la relació també s'omple la llista amb les entitats associades del costat *n* de la relació. Si aquesta llista no és massa grossa la penalització no és important. Ara, si la llista és grossa el temps i la memòria necessaris per mantenir-la pot arribar a penalitzar el rendiment. A més pot ser omplim la llista i no l'utilitzem. Una altra cosa a considerar és que si cada entitat de la llista té per la seva part altres llistes associades, també les omplirem, i així successivament. Es podria donar el cas de recuperar una sola entitat i haver de llegir la base de dades completa!

Lazy: Mandrós. La llista amb les entitats del costat *n* de la relació es carreguen en accedir a la llista des de l'aplicació. Només es carrega si és necessari.

- Els valors per defecte són:
  - OneToMany: LAZY
  - ManyToOne: EAGER
  - ManyToMany: LAZY
  - OneToOne: EAGER

Claus primàries compostes. Tipus incrustats.

- @Embeddable: Indica que la classe forma part d'una Entitat i que els seus atributs es maparan a la taula referenciada per aquella classe.
- @Embedded: Marca un atribut d'una entitat com a incrustat. El tipus de l'atribut ha de ser d'una classe marcada amb l'anotació @Embeddable.
- @EmbeddedId: Marca un atribut d'una entitat com a incrustat i identificador. El tipus de l'atribut ha de ser d'una classe marcada amb l'anotació @Embeddable. És sol utilitzar per a claus primàries compostes.

Les taules amb clau primaria composta per més d'un atribut tenen un tractament una mica especial. Es crea una classe nova amb els camps de la clau primària anotada amb *@Embeddable* i dins l'entitat s'afegeix un atribut d'aquesta classe marcat amb *@EmbeddedId*. La raó és per poder comparar dos objectes d'aquesta entitat més fàcilment.

Les classes anotades amb @Embeddable han d'implementar la interfície Serializable i els mètodes *equals* i *hashCode*.

Claus primàries compostes. @IdClass.

Una altra manera d'implementar una clau primària composta és utilitzant @IdClass.

Cream una classe amb atributs per als elements de la clau composta. Anotam cada un d'ells amb @Id i @Column

```
public class LliAutEntPK implements Serializable {
    @Id
    @Column(name = "FK_IDLLIB", nullable = false, insertable=false, updatable = false)
    private int fkIdllib;
    @Id
    @Column(name = "FK_IDAUT", nullable = false, insertable=false, updatable = false)
    private int fkIdaut;
```

sdfsd

A la classe d'entitat duplicam els atributs de la clau primària, amb les mateixes anotacions.

Decoram la classe amb l'anotació *@IdClass* especificant quina classe implementa la clau primària.

```
@Entity
@Table(name = "LLI_AUT", schema = "biblioteca", catalog = "")
@IdClass(LliAutEntPK.class)
public class LliAutEnt {
    @Id
    @Column(name = "FK_IDLLIB", nullable = false, insertable=false, updatable = false)
    private int fkIdllib;
    @Id
    @Column(name = "FK_IDAUT", nullable = false, insertable=false, updatable = false)
    private int fkIdaut;
```

Amb aquesta solució tenim les dades directament accessibles des de la classe d'entitat, sense haver de passar per la classe que representa la clau primària.

En canvi, si necessitam utilitzar la clau primària com un objecte és més fàcil amb *@EmbeddedId*.

Context de persistència: EntityManager

El terme context de persistència podríem dir que fa referència a l'entorn on es crea una entitat recuperant-la de la base de dades, modificant les seves dades, ... És el context dintre del qual es relaciona un objecte d'una classe entitat amb les corresponents dades de la base de dades.

És important tenir-ho en compte perquè, excepte per a la creació d'una entitat i realitzar el corresponent *Insert* a la base de dades, totes les altres operacions que involucrin l'entitat i la base de dades s'han de realitzar dins del mateix context de persistència.

L'objecte JPA que gestiona el context és l'*EntityManager*. Aquest objecte és el que proporciona els mètodes per modificar la base de dades des dels objectes Java.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("<nom de la UP>");
EntityManager em = emf.createEntityManager();
```

Aquest ***EntityManager*, entre d'altres coses manté una connexió amb la base de dades. Per tant, s'ha de mantenir actiu el mínim temps possible**.

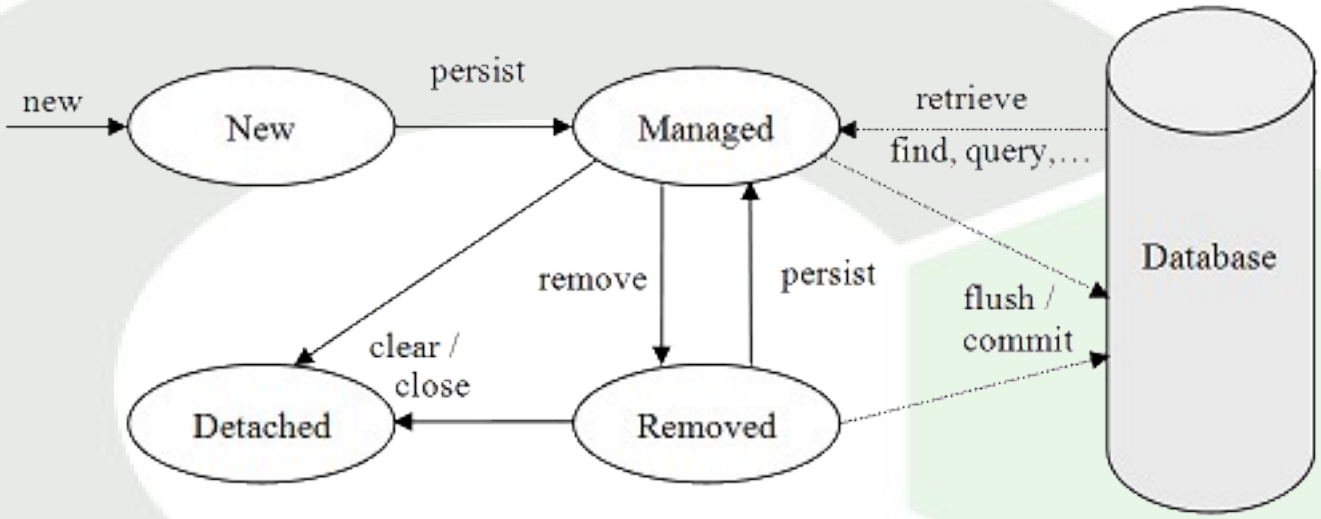
En aplicacions JSE és l'aplicació l'encarregada de crear *l'EntityManager*, tancar-lo, ... Normalment les operacions que afecten a la base de dades s'han de realitzar dins d'una transacció.

Per tancar *l'EntityManager* s'ha de tancar *l'EntityManagerFactory*. Si tancam directament *l'EntityManager* el programa no acaba.

```
em.close();

emf.close();
```

Cicle de vida d'una entitat



Al gràfic anterior es poden veure els estats per els que pot passar un objecte anotat amb @Entity:

- *New*: Acabam de crear l'objecte. Encara només resideix a memòria, no està relacionat amb la base de dades
- *Managed*: L'objecte en memòria està relacionat amb la base de dades. S'arriba a aquest estat quan el recuperam des de la base de dades o quan l'associam una altra vegada amb *persist* o *merge*.
- *Removed*: Hem indicat a *l'EntityManager* que volem esborrar l'objecte de la base de dades.
- *Detached*: L'objecte ha perdut la relació amb el context de persistència, per exemple perquè s'ha tancat l'objecte *EntityManager*.

Operacions: Mètodes d'EntityManager

L'*EntityManager* és l'objecte que gestiona la relació entre els objectes i la base de dades. Alguns dels seus mètodes són:

- .persist(Entitat): Relaciona un objecte nou o esborrat amb el context de persistència. Fa l'insert a la base de dades. Quan es faci un *commit* es farà permanent.
- .merge(Entitat): Torna a relacionar una entitat amb el context de persistència. Li passam l'entitat com a argument i torna una altra entitat que és la que està dins el context de persistència.
- .remove(Entitat): Elimina l'entitat del context de persistència. Fa un delete a la base de dades.
- .flush(): Actualitza la base de dades amb les entitats del context de persistència, sense fer cap commit ni rollback.
- .clear(): Elimina tots els objectes del context de persistència i els fa passar a l'estat Detached.

- .close(): Tanca l'EntityManager quan es gestionat per l'aplicació.
- .contains(Entitat): Rep una entitat com a argument i comprova si està associada o no al context de persistència de l'EntityManager.
- .detach(Entitat): Elimina l'entitat rebuda com a argument del context de persistència. Els canvis que hagués tengut l'entitat i que encara no s'haguessin passat a la base de dades no s'hi passaran.
- .find(Entity.class, clauPrimaria): Torna una entitat amb els valors retornats per la base de dades en fer el select amb la clauPrimaria que indicam.
- .refresh(Entitat): actualitza les dades de l'objecte Entitat amb les corresponents a la base de dades.

Transaccions

Les aplicacions *JSE* sempre les ha de gestionar les transaccions. Per fer-ho hem d'obtenir un objecte *EntityTransaction* de *l'EntityManager* amb el mètode *getTransaction*(). Els mètodes d'aquest objecte són:

- .begin(): Comença una transacció.
- .commit(): Acaba la transacció fent permanents els canvis a la base de dades.
- .rollback(): Acaba la transacció desfent els canvis a la base de dades.
- .setRollbackOnly(): Defineix la transacció de manera que només pot acabar amb un rollback.
- .getRollbackOnly(): Torna el valor d'aquesta propietat de la transacció.

El mateix objecte *EntityTransaction* es pot utilitzar per realitzar diverses transaccions successives.

Java Persistence Query Language

Aquest llenguatge és molt similar a l'SQL, només que **les consultes es fan en termes dels objectes Entitat que utilitza l'aplicació i no sobre les taules de la base de dades**.

Per exemple:

SQL: Select * from Especialitats

JPQL: Select e from Especialitat e

Les paraules clau de JPQL no són *case sensitive*. Els noms de les classes i dels seus atributs si.

El format de les sentències JPQL és calcat al de les sentències SQL. La majoria d'operadors també són els mateixos.

Select

La forma general del select, les parts entre [] són opcionals, és:

JPQL_statement ::= select_clause from_clause

[where_clause]

[groupby_clause]

[having_clause]

[orderby_clause]

El normal és definir un àlias per l'entitat i utilitzar-lo per fer referència a cada objecte que torna el select.

```
Select c from Cos c
```

De forma abreujada es pot posar com:

```
from Cos
```

La clàusula SELECT

Per a cada fila podem tornar un objecte sencer, un atribut de l'objecte, o una combinació d'atributs. En aquest últim cas hem de crear una classe amb els atributs que recuperarem i al select hem de crear un objecte d'aquesta classe. Es pot utilitzar *distinct* per evitar duplicats.

- Recuperar els objectes sencers

```
select e from Especialitat e
```

- Recuperar un atribut determinat

```
Select  distinct e.descripcio from Especialitat e
```

- Recuperar un atribut d'un atribut compost

```
select e.cos.id from Especialitat e
```

- Recupera una sèrie d'atributs utilitzant una classe auxiliar

```
select new org.spaad.Auxiliar(e.cos.id, e.descripcio) from Especialitat e
```

On Auxiliar és una classe que hem creat per recollir aquestes dades.

La clàusula FROM

- Una taula

```
select c from Cos c
```

- Inner Join

```
select c, e from Cos c JOIN c.especialitats e
```

- Left Outer Join

```
select c, e  from Cos c JOIN c.especialitats e
```

- Condicions al join

```
select c, e from Cos c JOIN c.especialitats e ON e.descripcio='INFORMATICA'
```

- Joins implícits

```
select e from Especialitat e where e.cos.id='0590'
```

La clàusula WHERE

Podem utilitzar

- Operadors relacionals: =, <>, <=, <, >=, >

```
select e from Especialitat e where descripcio='FILOSOFIA'
```

- BETWEEN AND

```
select e from Especialitat e where e.descripcio BETWEEN 'A' and 'D'
```

- Like

```
select e from Especialitat e where e.descripcio like  '%MAT%'
```

- Is null, is not null

- IN

```
select e from Cos c join c.especialitats e where c.idCos IN ['0590', '0591']
```

Per col·leccions

- Is Empty

```
select c from Cos c where c.especialitats is empty
```

- size

```
select c from Cos c where size(c.especialitats)>10
```

- Member of

```
select e from Especialitat e where a member of cos.especialitats
```

Agrupacions: clàusules GROUP BY i HAVING

- Podem agrupar els resultats. Els camps de la clàusula select que no formin part de l'agrupació han d'estar afectats per una funció d'agregació.

```
Select e.cos, count(e) from Especialitat e group by e.cos
```

Una alternativa pensant més en orientació a objectes

```
select c, c.especialitats.size as tamany from Cos c
```

- Podem filtrar els resultats agrupats amb Having

```
Select e.cos, count(e) from Especialitat e group by e.cos
having e.cos.idCos < '0592'
```

La clàusula ORDER BY

Podem ordenar els resultats per un o més camps, i, per a cada un d'ells, establir l'ordenació ascendent o descendent.

- Per defecte l'ordenació és ascendent.

```
Select e from Especialitat e order by e.cos.idCos
```

- Si volem l'ordenació descendent l'hem d'especificar.

```
Select e from Especialitat e order by e.cos.idCos desc
```

- Podem establir més d'un criteri d'ordenació, separats per comes

```
Select e from Especialitat e order by e.cos.id desc, e.descripcio
```

Funcions

Podem utilitzar les següents funcions entre d'altres:

- upper(String s): Transforma la cadena a majúscules.
- lower(String s): Transforma la cadena a minúscules.
- current_date(): Torna la data actual de la base de dades.
- current_time(): Torna l'hora actual de la base de dades.
- current_timestamp(): Torna la data i hora actuals de la base de dades.
- substring(String s, int offset, int length): Torna una subcadena de s, començant a la posició *ofsset* i de *length* caràcters.
- trim(String s): Elimina els espais en blanc del principi i del final de la cadena.

- length(String s): Torna la longitud de la cadena.
- locate(String search, String s, int offset): Torna la posició de la cadena search dins s. La cerca comença a partir de la posició *offset.*
- abs(Numeric n): Torna el valor absolut del paràmetre.
- sqrt(Numeric n): Torna l'arrel quadrada del paràmetre.
- mod(Numeric dividend, Numeric divisor): Torna el mòdul de la divisió.
- size(c): Torna la longitud de la col·lecció passada com a paràmetre.
- index(orderedCollection): Torna la posició de l'element dins d'una col·lecció ordenada. L'atribut ha d'estar anotat amb @OrderedCollection.

Funcions d'agregació

Podem utilitzar les següents funcions d'agregació entre d'altres. Totes s'apliquen només als objectes tornats per la consulta:

- AVG: Torna la mitjana de l'atribut passat com a paràmetre.
- COUNT: Torna la quantitat d'objectes que torna la consulta.
- MAX: Torna el màxim de l'atribut passat com a paràmetre.
- MIN: Torna la mínim de l'atribut passat com a paràmetre.
- SUM: Torna la suma de l'atribut passat com a paràmetre.

Update

La forma general de l'update, les parts entre [] són opcionals, és:

update_statement :: = update_clause [where_clause]

Les modificacions d'un objecte en concret es fan utilitzant els setters de l'objecte. La instrucció *update* s'ha d'utilitzar només per actualitzacions massives.

Funciona només sobre una classe i permet modificar un o més atributs de cada objecte que torni la condició del *where*.

No permeten *JOINS* però es poden seguir les relacions a través dels atributs que les representen. No es poden actualitzar les relacions on la classe de l'update sigui el costat 1 d'una 1 a n.

Els *update* tornen el nombre de columnes modificades a la base de dades.

```
UPDATE Employee e SET e.salary = 60000 WHERE e.salary = 50000
```

Aquest exemple modifica el salari de tots els empleats que el tenen establert a 50000 amb el nou valor de 60000.

Delete

La forma general del delete, les parts entre [] són opcionals, és:

delete_statement :: = delete_clause [where_clause]

L'eliminacions d'un objecte en concret es fa utilitzant el mètode *remove* de l'*EntityManager*. La instrucció *delete* s'ha d'utilitzar només per eliminacions massives.

Funciona només sobre una classe i permet eliminar els objectes que torni la condició del *where*.

No permet *JOINS* però es poden seguir les relacions a través dels atributs que les representen.

Els *delete* tornen el nombre de columnes modificades a la base de dades.

```
DELETE FROM Employee e WHERE e.department IS NULL
```

Aquest exemple elimina tots els empleats sense departament.

JPQL a l'aplicació

Per executar una consulta jpql dins l'aplicació depenem de l'*EntityManager*. Aquesta classe té un mètode sobrecarregat, *createQuery*.

La primera versió té un únic paràmetre, la cadena de text amb la consulta. Torna un objecte de tipus *Query*. Els resultats que torni aquesta query seran de tipus *Object*. Per tant, en recuperar-los, haurem de fer el càsting.

```
Query query=em.createQuery("SELECT c FROM Cos c");
```

La segona versió té dos paràmetres, el primer és la cadena amb la consulta, i el segon la classe dels resultats que torna. Torna un objecte de tipus *TypedQuery* parametritzat amb la classe que hem passat al mètode *createQuery*. L'avantatge és que no farà falta fer el càsting en recuperar els resultats.

```
TypedQuery<Cos> query=em.createQuery("SELECT c FROM Cos c", Cos.class);
```

El tipus que retorna la consulta depèn de com s'hagi fet:

- Si recupera un sol objecte d'una classe, torna un objecte d'aquesta classe

```
TypedQuery<Cos> q= em.createQuery("select e from Cos e", Cos.class)
```

- Si és un atribut, el tipus de l'atribut

```
TypedQuery<String> q= em.createQuery("Select c.descripcio from Cos c",
String.class);
```

- Recupera una sèrie d'atributs utilitzant una classe auxiliar

```
TypedQuery<String> q= em.createQuery("select new org.spaad.Auxiliar(e.cos.id,
e.descripcio) from Especialitat e", Auxiliar.class);
```

- Recuperar diversos objectes: En aquest cas s'ha d'utilitzar una *Query*. Cada resultat que torna és un *Object[]*. Cada posició d'aquest array s'ha d'aplicar el casting corresponent.

```
Query query = em.createQuery("select c, e from Cos c JOIN c.especialitats e");
List<Object[]> resultList = query.getResultList();
for(Object[] fila : resultList){
   Cos c=(cos)fila[0];
   Especialitat e=fila[1];
```

Paràmetres

Podem parametritzar la consulta modificant-la directament, però això la pot fer vulnerable a atacs d'injecció de codi.

Una altra manera de fer-ho és utilitzant paràmetres. JPQL permet dues formes de fer-ho:

- Per nom. Per declarar un paràmetre és posen dos punts i el nom del paràmetre. És la forma recomanada.

```
TypedQuery<Especialitat> query=em.createQuery("select e from Especialitat e where e.descripcio=:nom",Especialitat.class);
query.setParameter("nom","INFORMATICA");
```

- Per posició. Es posa un interrogant i el nombre del paràmetre.

```
TypedQuery<Especialitat> query=em.createQuery("select e from Especialitat e where e.descripcio=?1",Especialitat.class);
query.setParameter(1,"INFORMATICA");
```

Mètodes per consultes

Alguns dels mètodes que inclou **l'EntityManager** per a consultes són:

- .createQuery(jpql): Torna un objecte Query amb el jpql que rep com a paràmetre. Els resultats seran de tipus Object. Haurem de fer el càsting a mà.

```
Query query = em.createQuery("select e from Especialitat e");
```

- .createQuery(jpql,Tipus.class):Torna un objecte TypedQuery<Tipus> amb el jpql que rep com a paràmetre. Els resultats seran del tipus especificat.

```
TypedQuery<Cos> query = em.createQuery("select c from Cos c");
```

Mètodes per consultes

Alguns dels mètodes que inclou **Query** i **TypedQuery** són:

- .setParameter(nom o posició, valor): Estableix el valor del paràmetre de la consulta.
- .setFirstResult(int): Quina posició ocuparà el primer valor retornat. És a dir, si passam com a paràmetre 25, el primer valor retornat serà el que ocupa la posició 25 dins la llista de resultats.
- .setMaxResults(int): Quants valors tornarà com a màxim.
- .getSingleResult(): Per recuperar el resultat de querys que només tornen un únic valor.
- .getResultList(): Per recuperar la llista de resultats de querys que tornen més d'un valor.
- .executeUpdate(): Per *Updates* i *Deletes* massius.

Named Queries

Les querys que hem vist fins ara són dinàmiques, cada vegada que s'executa el *createQuery* s'han de compilar a l'SQL de la base de dades, amb el cost en temps d'execució que això implica.

JPA ens permet definir querys estàtiques a la definició de la classe. Aquestes querys es compilen a l'SQL de la base de dades una sola vegada en carregar la classe i, per tant, són més eficients. Es solen utilitzar per consultes molt freqüents.

El nom de la consulta és global a l'aplicació, per tant es sol posar com a prefixe el nom de la classe.

```
@NamedQuery(name="Especialitat.findAll","SELECT e FROM Especialitat e")
public class Especialitat {
```

Si hi ha més d'una *NamedQuery* es tanquen dins l'anotació *@NamedQueries*.

```
@NamedQueries({
        @NamedQuery(name = "Especialitat.findAll", query = "SELECT e FROM
Especialitat e"),
        @NamedQuery(name = "Especialitat.findByDescripcio", query = "SELECT e
        FROM Especialitat e where e.descripcio=:descripcio")
})
public class Especialitat {
```

Els mètodes de l'*EntityManager* per utilitzar-les són:

- .createNamedQuery(nomConsulta): Torna un objecte Query amb la NamedQuery que rep com a paràmetre. Els resultats seran de tipus Object.

```
Query query = em.createNamedQuery("Especialitat.findAll");
```

- .createNamedQuery(nomConsulta,Tipus.class):Torna un objecte TypedQuery amb la NamedQuery que rep com a paràmetre. Els resultats seran del tipus especificat.

```
TypedQuery<Especialitat> query = em.createNamedQuery("Especialitat.findAll",
Especialitat.class);
```