



Informàtica

ICB0 Desenvolupament d'aplicacions multiplataforma

M06 Accés a dades

UF2 Persistència en BDR-BDOR-BDOO

Diari d'activitats

Isidre Guixà

Curs 2024/25

| Què farem a la UF2? | 12/12/24 |
|--|-------------|
| <p>El programa oficial de la UF2 incorpora:</p> <ul style="list-style-type: none"> - Implementació de capes de persistència per a BDR-BDOR-BDOO - Eines de mapatge ORM <p>La programació de la UF2 en el M&F desestima destinar part de la UF2 en la implementació de capes de persistència per a BDR-BDOR-BDOO per dos motius:</p> <ul style="list-style-type: none"> - Amb el desenvolupament de capes de persistència efectuat a la UF3, capa de persistència via JDBC per BDR desenvolupada en el projecte1 i els coneixements de JDBC i BDOO obtinguts a UF6 de M03, queda coberta la implementació de capes de persistència per a BDR-BDOR-BDOO. - Les empreses demanden el coneixement d'eines ORM, fet que precisa d'un elevat nombre d'hores. <p>Per aquest motiu, en el M&F la UF2 es centra en l'accés a dades via eines ORM.</p> | |
| Eines de mapatge objecte-relacional (ORM) – Introducció - Dossier de l'IOC (només ORM) | 12-16/12/24 |
| <p> 2.1. Concepte de mapatge objecte-relacional 2.2. Eines de mapatge 2.2.1. Tècniques de mapatge 2.2.2. Llenguatge de consulta 2.2.3. Tècniques de sincronització (JDO, JPA 2.0 – JSR 317, JPA 2.1/2.2 - JSR 338) 2.3.3. Característiques generals del JPA - Javadocs oficials: JPA 2.0 – JSR 317, JPA 2.1/2.2 – JSR 338. </p> <p>Actualitat de les versions de JDO – JPA (no instal·leu cap de les versions que indica el dossier)</p> <p>L'agost de 2017 es publica JPA 2.2 com a revisió de JPA 2.1 dins el mateix JSR 338, del que podem veure un resum de les novetats aquí.</p> <p>El 2019 es va canviar el nom de JPA per Jakarta Persistence com a part de Jakarta EE, que és l'evolució <i>open source</i> de Java EE -propietat d'Oracle- i es publica la versió 3.0 el setembre de 2020. Aquest canvi inclou el canvi de nom de paquets i propietats de <code>javax.persistence</code> per <code>jakarta.persistence</code>.</p> <p>El 30/03/2022 es publica Jakarta Persistence 3.1, que precisa Java SE 11+.</p> <p>El 30/04/2024 es publica Jakarta Persistence 3.2, que és la darrera versió i que precisa Java SE 17+.</p> <p>Versions estables que suporten <i>Jakarta Persistence 3.1</i> existents en el moment de començar aquesta UF en el curs 24/25 i de les que en comprovarem el funcionament:</p> <ul style="list-style-type: none"> • Versió estable d'Hibernate: 6.6.3 (12/12/24) que requereix JDK11/17/21/22/23 • Versió estable d'EclipseLink: 4.0.4 (19/07/24) que requereix JDK11/17 <p>A data 12/12/24, les versions Jakarta 3.2 d'Hibernate i EclipseLink estan en desenvolupament (<i>Beta</i> i <i>Alfa</i>).</p> <p>Altres productes:</p> <ul style="list-style-type: none"> • DataNucleus (JPOX) és un producte OpenSource que compleix els 2 estàndards: <ul style="list-style-type: none"> - JAKARTA 3.1+ / Jakarta 3.0+ - JDO 3.2+ • ObjectDB, que suporta JPA i JDO. • Apache OpenJPA, que suporta JPA 2.2 i Jakarta Persistence API 3.0 <p>Molts productes, a més d'ORM, faciliten solucions per altres tipologies de SGBD!!!</p> <p>Productes que utilitzarem en el curs 24/25:</p> <ul style="list-style-type: none"> • Hibernate 6.6.3. S'aconsella crear a NetBeans la biblioteca amb nom Hibernate 6.6.3 ja que els projectes solució que es facilitaran en aquest dossier incorporaran una biblioteca amb aquest nom. Les llibreries corresponents a <code>hibernate-core-6.6.3.Final</code> es poden obtenir a partir d'aquest repository maven. A l'aula Classroom es facilita el conjunt de llibreries necessàries a incorporar a la biblioteca. El primer projecte <i>Hibernate</i> usarà biblioteca amb nom Hibernate 6.6.3 (Only Jakarta 3.1) que només ha de contenir la llibreria <code>jakarta.persistence-api-3.1.0.jar</code>. | |

- [EclipseLink 4.0.4](#). S'aconsella crear a NetBeans la biblioteca amb nom **EclipseLink 4.0.4** ja que els projectes solució que es facilitaran en aquest dossier incorporaran una biblioteca amb aquest nom. Aquesta biblioteca ha de contenir les llibreries següents de l'EclipseLink 4.0.4:
 - Llibreries de la carpeta jlib/jpa.
 - Llibreria jlib/eclipselink.jar
 El primer projecte *Eclipse* usarà biblioteca amb nom **EclipseLink 4.0.4 (Only Jakarta 3.1)** que només ha de contenir la llibreria jakarta.persistence-api.jar.
- En totes les biblioteques, aconsellable introduir el javadoc de *Jakarta Persistence 3.1* descarregable [aquí](#).

| | |
|---|-----------------|
| Peces bàsiques de JPA - Dossier de l'IOC (només ORM) | 16/12/24 |
|---|-----------------|

2.4.1. Entitats

2.4.2. El gestor d'entitats

2.4.3. Unitats de persistència

| | |
|---|-----------------|
| Definició d'Unitat de Persistència – Creació d'EntityManager | 18/12/24 |
|---|-----------------|

- La UP ha d'englobar el conjunt de propietats necessàries per configurar la connexió amb el SGBD i la funcionalitat de la persistència.
- La UP ha d'existir dins un fitxer de nom `persistence.xml` ubicat dins la carpeta META-INF a l'arrel del projecte. Aquest fitxer (que és únic) pot contenir més d'una unitat de persistència, cadascuna amb un nom diferent. Aquest nom és el que s'indica en el mètode per crear l'EntityManagerFactory:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory ("nomUP")
```

I dins la UP del fitxer `persistence.xml` hi ha les propietats necessàries per aconseguir la connexió amb el SGBD i també altres propietats referents al funcionament de la persistència.

- Les propietats de la UP definides dins el fitxer `persistence.xml` poden ser substituïdes/completades en la creació de l'EntityManagerFactory, utilitzant un mètode alternatiu de creació:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory ("nomUP", props)
```

on `props` és un conjunt de propietats que substitueix/completa les existents en el `persistence.xml`.

En algun dels projectes que desenvoluparem utilitzarem aquesta possibilitat.

Una vegada obtingut l'EntityManagerFactory cal obtenir l'EntityManager via `createEntityManager`.

Els entorns de desenvolupament, com per exemple NetBeans, acostumen a facilitar la possibilitat de crear l'arxiu de persistència, però potser no és el que correspon a la versió de JPA/Jakarta que estem usant.

Creació manual (nostre cas):

- Crear carpeta META-INF a l'arrel del projecte que posa en marxa l'aplicació.
- Crear dins un arxiu `persistence.xml` amb capçalera següent segons versió:

Capçalera del fitxer `persistence.xml` per JPA 2.0:

```
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
```

Capçalera del fitxer `persistence.xml` per JPA 2.1:

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

Capçalera del fitxer `persistence.xml` per JPA2.2: Apartat 8.3 d'[especificació JPA2.2](#)

```
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
```

Capçalera del fitxer `persistence.xml` per Jakarta 3.1: Apartat 8.3 d'[especificació Jakarta Persistence 3.1](#)

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd" version="3.0">
```

És el mateix esquema XML que per la versió 3.0 de *Jakarta Persistence*. Aquesta capçalera també està accessible dins el fitxer `persistence_3_0.xsd` adjunt a aquests materials.

Cal finalitzar el fitxer `persistence.xml` amb `</persistence>`. L'element `<persistence>` ha de contenir en el seu interior les unitats de persistència que calgui.

Compte amb NetBeans!!! En obrir un arxiu `persistence.xml`, NetBeans presenta un formulari per emplenar-lo però pot no ser adequat per la versió de Jakarta Persistence que anem a usar. Cal seleccionar la pestanya superior *Source* per veure el codi del fitxer i gestionar nosaltres directament el seu contingut.

⇒ El fitxer `persistence.xml` ha de validar l'esquema `persistence_3_0.xsd` adjunt.

Cada unitat de persistència és un node similar a:

```
<persistence-unit name="nom up" transaction-type="RESOURCE_LOCAL">
  <provider>classe JPA del fabricant</provider>
  <properties>
    <property name="jakarta.persistence.jdbc.url"
      value="URL jdbc per connectar amb SGBD"/>
    <property name="jakarta.persistence.jdbc.user" value="usuari"/>
    <property name="jakarta.persistence.jdbc.password" value="contrasenya"/>
    <property name="jakarta.persistence.jdbc.driver"
      value="classe JDBC"/>
    <!-- més propietats property -->
  </properties>
</persistence-unit>
```

Element `provider`

L'element `provider` dins la unitat de persistència ha de contenir el nom de la classe principal que implementa JPA, facilitada pel corresponent fabricant:

- Hibernate <= 5.1.x: `org.hibernate.ejb.HibernatePersistence`
- Hibernate >= 5.2.x i posteriors: `org.hibernate.jpa.HibernatePersistenceProvider`
- EclipseLink: `org.eclipse.persistence.jpa.PersistenceProvider`

Element `properties`

- Propietats que defineixen la connexió (driver JDBC, url JDBC, usuari, contrasenya,...)
- Dades per connectar amb Oracle (port habitual: 1521)
 - Driver: `ojdbc11.jar` (aula del Classroom)
 - URL: `jdbc:oracle:thin:@//IP:PORT/nomBD`
 - Classe JDBC: `oracle.jdbc.driver.OracleDriver`
- Dades per connectar amb PostgreSQL (port habitual: 5432)
 - Driver: `postgresql-42.2.5.jar` (aula del Classroom)
 - URL: `jdbc:postgresql://IP:PORT/nomBD`
 - Classe JDBC: `org.postgresql.Driver`



- Dades per connectar amb MariaDB (port habitual: 3306)
Driver: mariadb-java-client-3.3.2 (aula del Classroom)
URL: jdbc:mariadb://IP:PORT/nomBD
Classe JDBC: org.mariadb.jdbc.Driver
- Dades per connectar amb MySQL8+ (port habitual: 3306)
Driver: mysql-connector-java-8.0.13 (aula del Classroom)
URL: jdbc:mysql://IP:PORT/nomBD
Classe JDBC: com.mysql.cj.jdbc.Driver
- Dades per connectar amb MySQL5 (port habitual: 3306)
Driver: mysql-connector-java-5.1.41-bin_JDJ8-7-6 (aula del Classroom)
URL: jdbc:mysql://IP:PORT/nomBD
Classe JDBC: com.mysql.jdbc.Driver
- **Compte en MySQL8:** Si apareix error java.sql.SQLException: The server timezone value ... is unrecognized or represents more than one timezone completar la URL com segueix:
jdbc:mysql://IP:PORT/nomBD?serverTimezone=UTC

Els projectes solució que es facilitaran en aquest dossier incorporaran biblioteques amb noms:

- **JDBC Oracle**, amb la llibreria adequada JDBC per connectar amb l'Oracle que correspongui
- **JDBC PostgreSQL**, amb la llibreria adequada JDBC per connectar amb el PostgreSQL que correspongui
- **JDBC MariaDB**, amb la llibreria adequada JDBC per connectar amb MariaDB que correspongui
- Propietats específiques de cada proveïdor, com:
 - [hibernate.dialect](#), per informar a Hibernate el "dialecte SQL" adequat per comunicar amb el SGBDR; si un projecte no té aquesta propietat i hi ha algun problema que no permet la connexió (per exemple contrasenya errònia), Hibernate no informa correctament de la causa del problema per què no té el "dialecte SQL" informat. A més a més, en cas que Hibernate hagi de procedir a la creació o modificació de taules (ho veurem més endavant), si el "dialecte SQL" no és l'adequat, pot no generar adequadament les sentències. El valor a usar ha de ser una de les classes incorporades en el paquet [org.hibernate.dialect](#). Exemple:
 - Oracle11gR2+: org.hibernate.dialect.OracleDialect
 - PostgreSQL11+: org.hibernate.dialect.PostgreSQLDialect
 - MySQL5.7+: org.hibernate.dialect.MySQLDialect
 - MariaDB10.3+: org.hibernate.dialect.MariaDBDialect
- Altres que anirem incorporant a mida que es necessitin

Esquelet de programa JPA

El projecte 241218_0_EsquemaProgramaJPA conté l'esquelet d'una aplicació JPA:

- Crea una EntityManagerFactory a partir d'una UP indicada.
- Crea un EntityManager que gestiona la connexió amb un SGBDR.
- Aplicació en funcionament.
- Abans de finalitzar aplicació:
 - o Si hi ha EntityManager, tancar-lo previ rollback de transaccions obertes.
 - o Si hi ha EntityManagerFactory, tancar-lo.

Observar:

- No hi ha META-INF. Aquest, amb UP que correspongui, ha de residir en projecte/aplicació que executa.
- Només conté llibreria amb la definició de JAKARTA 3.1 (la d'Hibernate o d'Eclipse –no hi ha genèrica-) per poder incorporar els import de Persistence, EntityManagerFactory i EntityManager.

A continuació comprovarem l'execució d'aquest programa en 6 variants:

- Hibernate – EclipseLink
- SGBD Oracle – PostgreSQL - MariaDB

Per què fer rollback de transaccions obertes abans de tancar EntityManager?

- L'API JDBC defineix:

It is strongly recommended that an application explicitly commits or rolls back an active transaction prior to calling the close method. If the close method is called and there is an active transaction, the results are implementation-defined.

És a dir, si es tanca la connexió amb transaccions actives, que succeeixi commit o rollback depèn de l'actuació del connector JDBC emprat.

- Oracle, en la documentació del seu connector `ojdbc8.jar`, diu:

If the auto-commit mode is disabled and you close the connection without explicitly committing or rolling back your last changes, then an implicit COMMIT operation is run.

- [SQLServer](#), en la documentació diu:

Calling the close method in the middle of a transaction causes the transaction to be rolled back.

- PostgreSQL & MySQL, via comprovació (no trobat en documentació), executen `rollback` en tancar connexió

Davant el fet que no tots els SGBD actuen de la mateixa manera, és altament recomanable que abans de tancar una connexió (tancar `EntityManager` en JPA), es revisi si hi ha una transacció activa i, en cas afirmatiu, sembla lògic forçar un `rollback`:

```
if (em.getTransaction().isActive()) {  
    em.getTransaction().rollback();  
}
```

Informació referent a les Unitats de Persistència dels 6 projectes de comprovació:

El fitxer `persistence.xml` incorpora tres UP:

- UP-Oracle, per connectar amb un Oracle de la VM
- UP-PostgreSQL, per connectar amb un PostgreSQL 15 (aules del Milà)
- UP-MariaDB, per connectar amb un MariaDB 10 (XAMP de les aules del Milà)

Si voleu connectar amb SGBD diferents, tingueu en compte dades a canviar.

Canvieu també, evidentment, dades de connexió com IP, BD, usuari i contrasenya.

Comprovació 1: Projectes

- 241218_1_JPA_HibernateNomesLibJPA
- 241218_2_JPA_EclipseLinkNomesLibJPA

Els dos projectes només incorporen la llibreria corresponent a l'API Jakarta 3.1, és a dir, NO la implementació d'Hibernate o d'EclipseLink.

- En el cas d'Hibernate, la llibreria **Hibernate 6.6.3 (Only Jakarta 3.1)**.
- En el cas d'EclipseLink, la llibreria **EclipseLink 4.0.4 (Only Jakarta 3.1)**.

En ambdós casos i en qualsevol SGBD, el programa peta per què no hi ha accés a la llibreria del corresponent proveïdor ORM (Hibernate / EclipseLink). No arriba a crear l'`EntityManagerFactory`.

Comprovació 2: Projectes

- 241218_3_JPA_HibernateJPAComplet amb llibreria **Hibernate 6.6.3**.
- 241218_4_JPA_EclipseLinkJPAComplet amb llibreria **EclipseLink 4.0.4**

Els dos projectes incorporen les llibreries del corresponent proveïdor ORM (Hibernate / EclipseLink). En teoria, la seva execució hauria de crear la factoria (`EntityManagerFactory`) doncs ja es té accés al proveïdor.

- En Hibernate no té lloc la situació teòrica... En crear la factoria, peta amb error:
Unable to load class [...la que correspongui segons SGBD...]

L'error diu que no troba la classe JDBC per connectar, i no arriba a crear la factoria (no surt el missatge). En *Comprovació 3*, quan ja hi haurà el driver JDBC, comprovarem que crea la factoria. És a dir, Hibernate crea la factoria el moment d'establir connexió, quan es crea l'`EntityManager`.

- En EclipseLink sí té lloc la situació teòrica i la factoria es crea (surt missatge) abans d'intent d'establir connexió

Comprovació 3: Projectes

- 241218_5_JPA_HibernateJPA+JDBC

Observem que la UP incorpora la propietat `hibernate.dialect` (no imprescindible però recomanable).

- 241218_6_JPA_EclipseLinkJPA+JDBC

Els dos projectes incorporen els connectors JDBC per als 3 SGBDR i els programes ja s'executen en qualsevol dels 3 SGBD sempre i quan puguin establir la connexió amb el corresponent SGBD segons informació en la UP.

Si tot va bé, apareixeran els missatges:

```

EntityManagerFactory creada
EntityManager creat
EntityManager tancat
EntityManagerFactory tancada
  
```

Hibernate dona molts missatges informatius en vermell, que també apareixen per la consola. Normal. Feu una ullada al contingut. *EclipseLink* també treu informació, en menor quantitat i en negre.

La quantitat d'informació d'aquestes eines acostuma a ser configurable via propietats específiques de les eines.

Com funciona JPA?

18/12/24

A l'apartat 2.2.1 del dossier IOC s'explica la tècnica que utilitzen les diferents tipologies d'eines ORM. En qualsevol cas es tracta de poder informar a l'eina ORM de les classes que son persistents, és a dir, les classes de les que podem tenir objectes que interressi emmagatzemar en un SGBDR.

Per tant, en una aplicació conviuran:

- Classes no persistents: cap dels seus objectes necessita ser emmagatzemat en una BD
- Classes persistents: algun dels seus objectes (no tots) necessiten ser emmagatzemats en una BD

En el cas de JPA:

- Per informar de les classes que són persistents, JPA facilita dos mecanismes:
 - "Marcar" les classes persistents amb anotacions (similar al marcatge de JAXB)
 - Introduir les "marques" en arxiu `xml`, sense tocar el codi de la classe
 Poden conviure els dos mecanismes i, si per una classe existeix mapatge XML i mapatge via anotacions, el mapatge XML preval sobre les anotacions. Això permet que les anotacions JPA que pugui contenir una classe en el seu codi siguin sobreescrites per marques diferents via XML
- Per informar quins objectes d'una classe persistent han de passar a ser "entitats" gestionades per JPA (és a dir, objectes emmagatzemats a la BD), el programador disposa de mètodes en l'`EntityManager`.

Independentment del tipus de mapatge (anotacions o XML), hem de tenir present que l'eina ORM fa la traducció:

objecte de classe ⇔ fila de taula

tot i que en alguna ocasió, les dades d'un objecte pot quedar emmagatzemades en taules diferents:

objecte de classe ⇔ files de taules

i respecte les taules de la BD hem de saber que l'eina ORM pot estar configurada per a que, en posar-se en marxa, respecte les classes "marcades" com a persistents:

- Creï les taules corresponents si encara no existeixin.
- Comprovi si les taules corresponents coincideixen amb el "marcatge"
 - Si no coincideixen, executi `alter table` per aconseguir la coincidència
 - Si no coincideixen, generi excepció.
- Elimini i creï de nou les taules

No totes les eines JPA faciliten les mateixes possibilitats. L'estàndard JPA marca unes possibilitats concretes (que veurem) però cada eina JPA les "aplica" a la seva manera i no hi ha total uniformitat.

En tots els projectes que desenvolupem al llarg de la UF, haurem de tenir en compte com interessa configurar l'eina (Hibernate o EclipseLink), segons vulguem que mantingui o no les dades i l'estructura de les taules a la BD.

Requeriments inicials per la classe - Capítol 2.1 de [documentació oficial de Jakarta 3.1](#)

19/12/24

- Obligatorietat de constructor sense paràmetres, públic o protegit
- `Serializable`
- No pot ser final ni tenir mètodes `getter-setter` final.
- Ha de tenir obligatòriament un(s) camp(s) que la identifiquin i han de ser **IMMUTABLES**.

| Configuració mínima a les classes per començar a fer proves - Vídeo | 19/12/24 |
|---|----------|
| <ul style="list-style-type: none"> • Primeres anotacions obligatòries <ul style="list-style-type: none"> - Nivell de classe: <code>@Entity</code>, per indicar que és una classe persistent - Nivell de dades: <code>@Id</code>, per indicar camp PK (més endavant ja s'explicarà com fer en claus compostes) • Desenvolupem una primera classe que contingui un camp de cada tipus de dada habitual en Java per veure com els diversos proveïdors JPA els mapen en els diversos SGBD. <p>Projectes amb la classe <code>ProvaTipusDades</code> amb configuració mínima:</p> <p>241219_1_ProvesTraduccióTipusDades_Hibernate 241219_2_ProvesTraduccióTipusDades_EclipseLink La versió per EclipseLink pot necessitar la <code>property eclipselink.canonicalmodel.subpackage</code> (situació estranya)</p> <ul style="list-style-type: none"> • La carpeta <code>META-INF</code> amb el fitxer <code>persistence.xml</code> ha de residir a l'arrel de l'aplicació que invoca la persistència (no en el projecte que contingui les classes). • Incorporem a la UP les classes que estan marcades, per a que el proveïdor de persistència les tingui en compte, en element <code><class></code> (en ubicació segons indiqui <code>persistence_3_0.xsd</code>) <p>És possible que algun proveïdor de persistència, sense tenir la informació de les classes marcades, incorpori directament aquelles que contenen alguna marca.</p> <ul style="list-style-type: none"> • Les UP subministrades contenen les propietats següents que inicialment deixem comentades: De JPA: <code>jakarta.persistence.schema-generation.database.action</code> Pròpia d'Hibernate: <code>hibernate.hbm2ddl.auto</code> Pròpia d'EclipseLink: <code>eclipselink.ddl-generation</code> <ul style="list-style-type: none"> • Comprovació 1: Executem programa de proves amb les propietats anteriors comentades. Mirem la BD. No veiem cap taula creada. Motiu: NO hem informat a la UP quina acció ha de dur a terme. • Comprovació 2: Activem la <code>property</code>: <code>jakarta.persistence.schema-generation.database.action</code> que permet opcions: <code>none</code>, <code>create</code>, <code>drop-and-create</code>, <code>drop</code> (cercar aquesta propietat en PDF de documentació oficial Jakarta 3.1) <p>Executem el programa de prova comprovant què passa segons proveu amb <code>create</code>, <code>drop-and-create</code>,... Mirem la taula que es crea a la BD.</p> <p>Alerta!!! Perfecte usar l'opció <code>create</code> en la primera ocasió si volem que creï les taules i <code>drop-and-create</code> en següents, mentre s'afina la definició de les anotacions, però en explotació: <code>none</code>!!!</p> <p>La majoria d'ocasions, la creació de la BD no l'efectua l'aplicació sinó que s'efectua prèviament via guions.</p> <p>Alguns proveïdors de persistència faciliten propietat alternativa amb més possibilitats:</p> <ul style="list-style-type: none"> - Hibernate: <code>hibernate.hbm2ddl.auto</code> amb els valors: <ul style="list-style-type: none"> ✓ <code>validate</code>: validate the schema, makes no changes to the database (en explotació) ✓ <code>update</code>: update the schema (en desenvolupament) => Per treballar a classe ✓ <code>create</code>: creates the schema, destroying previous data. ✓ <code>create-drop</code>: drop the schema at the end of the session. - EclipseLink: <code>eclipselink.ddl-generation</code> amb els valors: <ul style="list-style-type: none"> ✓ <code>create-tables</code> ✓ <code>create-or-extend-tables</code>: Similar a <code>update</code> d'Hibernate => Per treballar a classe ✓ <code>drop-and-create-tables</code> ✓ <code>none</code> • Comprovació 3: Desactivar la <code>property</code> <code>jakarta.persistence.schema-generation.database.action</code> i activar la <code>property</code> <code>hibernate.hbm2ddl.auto</code> en Hibernate o la <code>property</code> <code>eclipselink.ddl-generation</code> en EclipseLink. | |

Executem el programa prèvia eliminació de la taula a la BD, per veure què passa segons el valor de la propietat.

- En l'execució dels projectes anteriors, deixant que el proveïdor JPA creï la taula, observem:

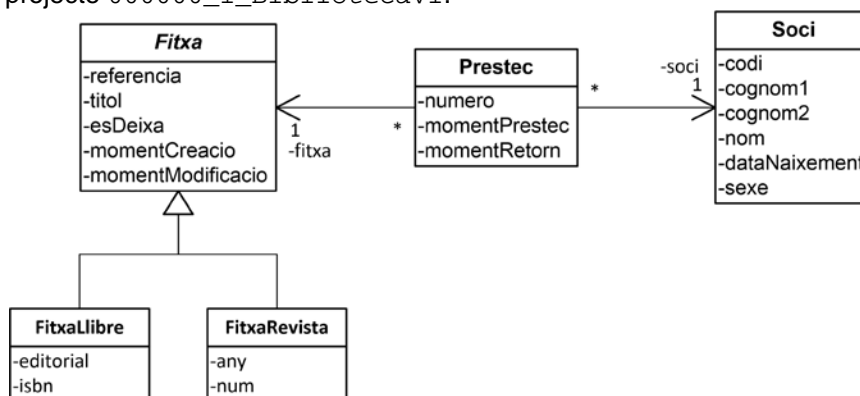
| Tipus de dada | Hibernate 6.6.3 | | | EclipseLink 4.0.4 | | |
|---------------|--------------------|------------------------|---------------|--------------------|------------------------|---------------|
| | Oracle 21c | PostgreSQL 15 | MariaDB 10.4 | Oracle 21c | PostgreSQL 15 | MariaDB 10.4 |
| Integer | NUMBER(10,0) | Integer | int(11) | NUMBER(10,0) | Integer | int(11) |
| BigDecimal | NUMBER(38,2) | numeric(38,2) | decimal(38,2) | NUMBER(38,0) | numeric(38,0) | decimal(38,0) |
| BigInteger | NUMBER(38,0) | numeric(38,0) | decimal(38,0) | NUMBER(38,0) | Bigint | bigint(20) |
| Boolean | NUMBER(1) | boolean | bit(1) | NUMBER(1) | Boolean | tinyint(1) |
| Byte | NUMBER(3) | smallint | tinyint(4) | NUMBER(3) | Smallint | tinyint(4) |
| Character | CHAR(1 BYTE) | character(1) | char(1) | CHAR(1 BYTE) | character(1) | char(1) |
| Double | FLOAT(24) | double precision | Double | NUMBER(19,4) | double precision | Double |
| Float | FLOAT(53) | real | Float | NUMBER(19,4) | double precision | Float |
| Long | NUMBER(19) | bigint | bigint(20) | NUMBER(19,0) | Bigint | bigint(20) |
| Short | NUMBER(5) | smallint | smallint(6) | NUMBER(5,0) | Smallint | smallint(6) |
| String | VARCHAR2(255 CHAR) | character varying(255) | varchar(255) | VARCHAR2(255 BYTE) | character varying(255) | varchar(255) |

Alerta: Donada la diversitat de tipus de dades que generen els proveïdors JPA en els diversos SGBD, la creació i/o modificació d'una BD en explotació s'acostuma a fer amb guions que incorporen les instruccions de creació i/o modificació adequades, amb possibles processos de preparació prèvia de dades i/o modificació posterior de dades. A més, en ocasions els tipus generats no són adequats (BigDecimal en cas d'EclipseLink 4.0.4)!!!

Pràctica

08/01/25

Es vol assolir i gestionar la persistència de les classes del següent UML, de les que ja es facilita una implementació en el projecte 000000_1_BibliotecaV1.



Pràctiques – Nomenclatura

08/01/25

Desenvoluparem 2 versions en simultani:

- Primer, amb anotacions a les pròpies classes, en projecte BibliotecaV1A que utilitzarem en projectes: BibliotecaV1A_Hibernate, utilitzant Hibernate com a proveïdor JPA
BibliotecaV1A_EclipseLink, utilitzant EclipseLink com a proveïdor JPA
- Segon, amb marques XML en fitxers externs, que utilitzarem en projectes: BibliotecaV1X_Hibernate, utilitzant Hibernate com a proveïdor JPA
BibliotecaV1X_EclipseLink, utilitzant EclipseLink com a proveïdor JPA
Aquests projectes sempre utilitzaran les classes de BibliotecaV1.

Quan es decideixi fer algun canvi en les classes de Biblioteca, passarem a nova versió BibliotecaV#.

Primeres anotacions – Primers mètodes per gestionar els objectes – Classe Soci

08/01/25

- Retocs a totes les classes per aconseguir els requeriments: constructor sense paràmetres, serializable, no final, mètodes setter-getter no final, camp(s) que hagi(n) de ser identificador(s) immutables.
- Anotacions a incorporar (només a Soci):
 - Nivell de classe: @Table
 - Nivell de dades: @Basic | @Column | @Temporal

- Respecte @Temporal, que admet valors DATE, TIMESTAMP (per defecte) i TIME, els projectes 250108_0_ProvesCampsTemporals_Hibernate i 250108_0_ProvesCampsTemporals_EclipseLink permeten comprovar el tipus de dada SQL que el proveïdor JPA usa per definir la columna en crear la taula.

La taula següent mostra els tipus usats per cada proveïdor en funció del SGBD.

| Tipus de dada Java | TemporalType | Hibernate 6.6.3 | | | EclipseLink 4.0.4 | | |
|--------------------|--------------|---|---------------|--------------|--------------------------|---------------|--------------|
| | | Oracle 21c | PostgreSQL 15 | MariaDB 10.4 | Oracle 21c | PostgreSQL 15 | MariaDB 10.4 |
| Date | DATE | date | date | date | date | date | date |
| Date | TIMESTAMP | timestamp(6) | timestamp(6) | datetime(6) | timestamp(6) | timestamp | datetime |
| Date | TIME | timestamp(6) | time(6) | time(6) | timestamp(6) | time | time |
| Calendar | DATE | Date | date | date | date | date | date |
| Calendar | TIMESTAMP | timestamp(6) | timestamp(6) | datetime(6) | timestamp(6) | timestamp | datetime |
| Calendar | TIME | timestamp(6) | time(6) | time(6) | timestamp(6) | time | time |
| GregorianCalendar | DATE | timestamp(6) | timestamp(6) | datetime(6) | date | date | date |
| GregorianCalendar | TIMESTAMP | timestamp(6) | timestamp(6) | datetime(6) | timestamp(6) | timestamp | datetime |
| GregorianCalendar | TIME | timestamp(6) | timestamp(6) | datetime(6) | timestamp(6) | time | time |
| LocalTime | DATE | no és possible marcar un camp LocalTime com a DATE | | | | | |
| LocalTime | TIMESTAMP | no és possible marcar un camp LocalTime com a TIMESTAMP | | | | | |
| LocalTime | TIME | timestamp(6) | time(6) | time(6) | EclipseLink no ho permet | | |

El tipus DATE d'Oracle emmagatzema data-temps, com TIMESTAMP, que a més permet fraccions de segon. Però en altres SGBDR com PostgreSQL i MariaDB, el camp DATE només emmagatzema dates. Per tant, caldrà usar TemporalType.DATE només quan els valors a emmagatzemar siguin dates sense temps i usarem TemporalType.TIMESTAMP per emmagatzemar moments temporals (data-temps).

- Usar majúscules o minúscules en els noms de taules, columnes, restriccions, índexs,...?
 - PostgreSQL** crea taules i columnes i índexs en **minúscules**, tant per les classes i camps pels que s'indiqui nou nom, com per les classes i camps pels que no s'indica cap nom específic per a la BD.
 - Oracle** crea taules i columnes i índexs en **majúscules**, tant per les classes i camps pels que s'indiqui nou nom, com per les classes i camps pels que no s'indica cap nom específic per a la BD.
 - MySQL/MariaDB** té un funcionament variable:
 - L'ús de majúscula/minúscula pels noms de les taules depèn del valor de la variable `lower_case_table_names` en el servidor (apartat `mysqld` de l'arxiu de configuració `my.ini`). [Aquí](#) explicació detallada dels valors possibles, que també depenen del tipus de S.O. En MariaDB 10.4.28 instal·lat en el XAMP per a Windows, està configurat amb el valor 2, fet que implica que els taules corresponents a les classes per les que no s'indica nom, quedaran enregistrades exactament igual al nom que s'indiqui en el marcatge o, si no s'indica nom, exactament igual que el nom de la classe. Potser millor configurar el servidor amb el valor 1 per a que sempre quedin en minúscules i assegurar uniformitat.
 - Els noms de les columnes són exactament igual al nom indicat en el marcatge o, si no s'indica nom, exactament igual que el nom del camp dins la classe. Donat que el nom dels camps en Java es fa en minúscules, potser millor usar sempre les minúscules pels noms en els marcatges efectuat.
- Diferència entre anotacions `optional=false` i `nullable=false`**

Segons wikibooks:

A Basic attribute can be optional if its value is allowed to be null. By default everything is assumed to be optional, except for an Id, which can not be optional. Optional is basically only a hint that applies to database schema generation, if the persistence provider is configured to generate the schema. It adds a NOT NULL constraint to the column if false. Some JPA providers also perform validation of the object for optional attributes, and will throw a validation error before writing to the database, but this is not required by the JPA specification. Optional is defined through the optional attribute of the Basic annotation or element.

Per altra banda, resposta d'un "gurú" de JPA a Stack Overflow:

The difference between optional and nullable is the scope at which they are evaluated. The definition of 'optional' talks about property and field values and suggests that this feature should be evaluated within the runtime 'nullable' is only in reference to database columns.

If an implementation chooses to implement optional then those properties should be evaluated in memory by the Persistence Provider and an exception raised before SQL is sent to the database otherwise when using 'updatable=false' 'optional' violations would never be reported.

Per últim, per Hibernate:

The Hibernate JPA implementation treats both options the same way in any case, so you may as well use only one of the annotations for this purpose.

- Què passa amb `optional/nullable` si un camp no es marca i JPA és qui crea el camp a la BD?

JPA actua amb els valors per defecte i, per tant, sembla que hauria de crear el camp sense la restricció `NOT NULL`. I així és pels camps que fan referència a objectes (`String`, `Integer`, `Long`, qualsevol classe...) però NO pels camps de tipus primitius, ja que un camp de tipus primitiu mai pot tenir valor `NULL` i, en conseqüència, per aquests camps, JPA els incorpora adequadament la restricció `NOT NULL` si no s'indica el contrari.

- Primers mètodes per gestionar dades persistents:

- `em.persist(obj)`

Marca l'objecte per fer-lo persistent (alta) i passa a ser controlat per l'Entity Manager. No passa a la BD. Si l'Entity Manager ja està controlant un objecte amb mateix ID, hauria de petar amb una `EntityExistsException`. Però la documentació JAKARTA 3.1 diu textualment:

If the entity already exists, the EntityExistsException may be thrown when the persist operation is invoked, or the EntityExistsException or another PersistenceException may be thrown at flush or commit time.

I aquí podem topar amb diferent funcionament segons proveïdor de persistència:

- ✓ Hibernate 6.6.3: Genera excepció en efectuar el marcatge (`persist`).
- ✓ EclipseLink 4.0.4: Genera excepció en traspasar els canvis a la BD (`flush` o `commit`)

- `em.flush()`

Enregistra els canvis a la BD (dins una transacció, sense fer `commit`; podríem fer `rollback`)

- `em.getTransaction.begin()`

Inicia transacció

- `em.getTransaction.commit()`

Tanca transacció activa validant canvis pendents a la BD. Peta si no transacció activa.

- `em.getTransaction.rollback()`

Tanca transacció activa sense validar canvis pendents a la BD. Peta si no transacció activa.

- `em.getTransaction.isActive()`

Informa si tenim transacció activa.

- `em.find(NomClasse.class, valorDeCampId)`

Recupera objecte persistent de la BD. Recupera `null` si no existeix.

- `em.createQuery` (llenguatge JPQL)

Per executar consultes similars a SQL, recuperant objectes!!!

Què passa quan es produeix alguna `PersistenceException`? Doncs que la transacció activa, si n'hi ha, queda marcada com a transacció de "només `rollback`" i si s'intenta fer `commit` es produirà una excepció. És responsabilitat del programa invocar `rollback`; del contrari, les modificacions passades a la BD pendents de validar (s'hagin fet via `flush`) poden ser validades => Compte!!! Veure següent apartat sobre actuació JDBC.

- `em.getTransaction.getRollbackOnly()`

Informa si la transacció activa està marcada de "només `rollback`". Peta si no transacció activa.

- `em.getTransaction.setRollbackOnly()`

Marca la transacció activa de "només `rollback`". Peta si no transacció activa.

En EclipseLink, els mètodes `commit`, `rollback`, `getRollbackOnly` i `setRollbackOnly` peten si no hi ha transacció activa (com marca JPA). Hibernate, però, no es queixa. Cal actuar com diu JPA i estar segurs de que hi ha transacció activa.

- Projectes desenvolupats:

250108_1_BibliotecaV1 (projecte "net" sense cap anotació i amb els requeriments mínims per a totes les classes: constructor sense paràmetres, `serializable`, no final, mètodes `setter-getter` no final, camp(s) que hagi(n) de ser identificador(s) immutables) .

250108_2_BibliotecaV1A (projecte anterior amb les anotacions per la classe `Soci`)

Observar que necessita, per compilar, el jar de JAKARTA 3.1 (indiferent d'Hibernate o EclipseLink)

Alerta amb l'atribut `columnDefinition` d'`@Column` per la columna `sexe` (veure apartat següent).

250108_3_BibliotecaV1A_Hibernate i 250108_4_BibliotecaV1A_EclipseLink, de proves amb marcatge via anotacions

Observar que necessiten incorporar llibreries JDBC (segons SGBD), el projecte BibliotecaV1A que conté les anotacions i llibreria Hibernate o EclipseLink (segons correspongui)

- Programa P02, que crea 2 socis (100 i 200), els fa persistents i finalment els enregistra a la BD.
- Si executeu P02 quan ja existeixen files amb codis 100 i 200 a la BD, s'observa que el programa peta en intentar fer el flush, que en aquest cas consisteix en intentar fer insert a la BD, però en fer el persist no peta per què per l'EntityManager són registres nous.
- Programa P03 que crea 2 socis 300 (300a i 300b amb mateix codi 300) i els vol fer persistents. *Hibernate* es queixa en intentar fer persist del segon, per què detecta que ja en té un en memòria. *EclipseLink* es queixa quan es fa el flush o commit.
- Programa P04 que recupera un objecte Soci de la BD (mètode find) i en modifica en nom, enregistrant els canvis.
 - Mètode find per recuperar un objecte coneixent el valor del(s) camp(s) @id.
 - Però per recuperar objectes (un o varis) que compleixin una determinada condició... llenguatge JPQL. (veure apartat 2.7 de [dossier de l'IOC](#))
- Programa P05 mostra com recuperar tots els objectes socis, via getResultList().
- Programa P06 mostra com recuperar un únic objecte, via getSingleResult(), tenint en compte que si no existeix o si n'hi ha més d'un, es genera excepcions.
- Programa P07 mostra com recuperar camps dels objectes, enlloc d'objectes, usant també getResultList() (també es pot usar getSingleResult()).

Respecte la utilització de columnDefinition...

L'anotació @Column incorpora la possibilitat d'usar columnDefinition per detallar com el SGBD ha de crear la columna dins la BD i, en cas d'emprar-lo, cal usar la sintaxi adequada al SGBD i, per desgràcia, la sintaxi de creació de columnes no és estàndard.

En cas d'usar-la sobre un camp que hagi de ser not null es pot optar per

- definir @Basic (optional=false) i @Column(nullable=false, columnDefinition="...") sense indicar not null dins columnDefinition o
- no definir @Basic (optional=false) i @Column (columnDefinition="... not null... ") indicant not null dins la definició de la columna,
- però NO es pot indicar @Basic (optional=false) i indicar not null dins columnDefinition per què al SGBD li arriba la notificació not null dues vegades, fet que és erroni.

La columna sexe de la classe Soci és candidat a usar columnDefinition per introduir la restricció relativa als valors que pot contenir la columna i, segons coneixements, sembla que la definició hauria de ser:

```
columnDefinition="char not null constraint SOCI_SEXE_CK check (sexe in ('M','F','?'))"
```

Doncs... aquesta definició és acceptada en Oracle i en PostgreSQL, però no en MariaDB 10.4.28, que admet:

```
columnDefinition="char not null check (sexe in ('M','F','?'))"
```

Conclusió: És adequat usar columnDefinition quan el marcatge s'efectua per un SGBDR concret, cas en el que podem tenir seguretat absoluta de la sintaxi a emprar. Com que els nostres marcatges volem que siguin vàlids per Oracle/PostgreSQL/MySQL/MariaDB, no usarem columnDefinition a no ser que tinguem plena seguretat que la definició sigui plenament estàndard. No cal oblidar que en un cas real, la creació de taules i índexs s'efectua via guió específic pel SGBDR a usar.

FYI sobre les restriccions CHECK en MySQL i MariaDB:

- [MySQL](#) no executa les restriccions CHECK en versions anteriors a 8.0.15; per compatibilitat amb altres SGBDR, les accepta però les ignora. A partir de versió 8.0.16, ja les accepta i executa.
- [MariaDB](#) no executa les restriccions CHECK en versions anteriors a 10.2.1; per compatibilitat amb altres SGBDR, les accepta però les ignora. A partir de versió 8.0.16, ja les accepta i executa.

Segons documentació, ha d'acceptar dues formes de definir la restricció:

CHECK(expression) given as part of a column definition

CONSTRAINT [constraint_name] CHECK (expression)



En cas d'usar columnDefinition, s'està indicant la instrucció en el moment de definir la columna i, en aquest cas, la sintaxi no permet definir el nom de la restricció.

Primeres marques amb XML – Classe Soci

15/01/25

[Vídeo del desenvolupament](#)

250115_1_BibliotecaV1X_Hibernate i 250115_2_BibliotecaV1X_EclipseLink, de proves amb marcatge XML.

Observar que la carpeta META-INF està SEMPRES en els projectes que creen EntityManager. En el cas de marcatge XML, els fitxers podrien estar fora META-INF, però acostumen a estar allí.

⇒ El fitxer XML de marcatge de cada classe ha de validar l'esquema `orm_3_1.xsd` adjunt.

Capçalera del fitxer XML per marcatge de classe per JPA 2.0:

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

Capçalera del fitxer XML per marcatge de classe per JPA 2.1:

```
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd" version="2.1">
```

Capçalera del fitxer XML per marcatge de classe per JPA2.2: **Capítol 12 de doc. oficial de JPA2.2 (12.3)**

```
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd" version="2.2">
```

Capçalera del fitxer XML per marcatge de classe per JAKARTA 3.1: **Capítol 12.3 de doc. oficial de Jakarta3.1**

```
<entity-mappings xmlns="https://jakarta.ee/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence/orm
https://jakarta.ee/xml/ns/persistence/orm/orm_3_1.xsd" version="3.1">
```

Aquesta capçalera també està informada dins el fitxer `orm_3_1.xsd` adjunt a aquests materials.

⇒ És fonamental que `persistence.xml` i els fitxers de mapatge siguin vàlids i si usem NetBeans per a la validació, en ocasions NetBeans informa que la validació ha finalitzat però amb un error previ. Teniu la solució en aquest [vídeo explicatiu sobre la validació dels fitxers XML \(persistence.xml i fitxer de mapatge\)](#)

En aquest primer projecte, estem utilitzant TOTS els SGBD i els 2 proveïdors JPA. Això és molt costós. Ho hem fet aquí per comprovar-ne el funcionament.

A partir d'ara, practicarem amb Hibernate en Oracle, doncs hem de centrar els esforços en JPA i no pas en les particularitats que ens podem trobar en diversos SGBD i diversos fabricants ORM.

Com controlar les instruccions SQL que executen els proveïdors de JPA

15/01/25

En els projectes anteriors hem incorporat ja algunes propietats específiques de cada proveïdor per poder veure les instruccions SQL que executen quan accedeixen a la BD. Explicació més detallada:

- EclipseLink – Més informació [aquí](#).
 - Per veure instruccions SQL (es veu tota la instrucció en una línia):

```
<property name="eclipselink.logging.level.sql" value="FINE"/>
```
 - Per veure els valors que es passen a les variables d'enllaç:

```
<property name="eclipselink.logging.parameters" value="true"/>
```
- Hibernate:
 - Per veure instruccions SQL (es veu tota la instrucció en una línia):

```
<property name="hibernate.show_sql" value="true"/>
```
 - Per veure la instrucció SQL formatada en varies línies (amb show previ a true):

```
<property name="hibernate.format_sql" value="true"/>
```




- Per veure els valors que es passen a les variables d'enllaç, és més complicat que en Eclipse. Cal seguir les explicacions de l'apartat *JDK Logger configuration...* d'aquest [enllaç](#) (cas per Hibernate 6+). Concretament, afegir al final del fitxer `logging.properties` que, en JDK17, es troba a `pathJDK/conf:`
Per Hibernate 6+
`java.util.logging.ConsoleHandler.level=FINEST`
`org.hibernate.level=INFO`
`org.hibernate.SQL.level=FINER`
`org.hibernate.orm.jdbc.bind.level=FINEST`
En cas d'afegir aquestes propietats, desactivar a `persistence.xml` la propietat `hibernate.show_sql`, doncs en cas de no fer-ho, mostra les instruccions duplicades.

Continuació pràctica – Classe Fitxa [Vídeo de la solució](#)

15/01/25

- Retocar la classe `Fitxa` per a que no sigui abstracta (oblideu-vos de `FitxaRevista` i `FitxaLlibre`)
- Fer la classe persistent, via anotacions i via marcatge XML
- Requeriments a nivell de taula:
 - o Un índex per títol
 - o Les fitxes s'identifiquen per la seva referència
 - o Llargada màxima de títol: 60 caràcters
- Comprovar creació de taula en ambdues vies
- Fer alguns programes que juguin amb objectes `Fitxa` (per això no pot ser abstracta)

Projectes solució:

- En primer lloc comprovem que la classe `Fitxa` ja té els requeriments mínims per poder-la fer persistent (`Serializable`, constructor sense paràmetres, NO final, getter-setter no final, camps id. immutables)
- 250115_3_BibliotecaV1
- 250115_4_BibliotecaV1A
- 250115_5_BibliotecaV1A_Hibernate
- 250115_6_BibliotecaV1X_Hibernate

Observacions a tenir en compte respecte les claus foranes

Les relacions entre classes (que començarem a veure a continuació) impliquen l'aparició de relacions entre les corresponents taules, les quals es defineixen amb claus foranes, amb estructura similar a:

```
FOREIGN KEY ( <COLUMN expression> {, <COLUMN expression>}... )
REFERENCES <TABLE identifier> [
    (<COLUMN expression> {, <COLUMN expression>}... ) ]
[ ON UPDATE <referential action> ]
[ ON DELETE <referential action> ]
```

El més habitual (i segur) és que l'estructura de la BD es creï via un guió i que no sigui el motor de persistència l'encarregat de crear-la.

Però, pel cas en que el motor de persistència hagi de crear l'estructura, ens convé tenir en compte alguns problemes (errors) que presenten Hibernate i EclipseLink.

En els temes següents veurem com es declaren relacions entre taules, en les que és normal que es vulgui prendre decisions respecte el nom de la clau forana (del contrari, el nom que JPA assigna no acostuma a donar cap informació sobre la FK) i les accions referencials a aplicar (on delete i on update) i, per aconseguir-ho, JPA facilita marcatge `@ForeignKey` | `foreign-key` amb atribut `name` per batejar la clau forana i atribut optatiu `foreignKeyDefinition` | `foreign-key-definition` per establir la definició de la clau forana.

Els problemes detectats no coincideixen en Hibernate i EclipseLink i sembla que EclipseLink segueix més l'estàndard definit per JPA que no pas Hibernate. Problemes:

- Poden no crear la FK amb el nom indicat en el marcatge.
- Poden no fer cas de la definició i, per tant, no incorporar les clàusules `on delete` i/o `on update` indicades.

En la versió que estem usant d'Hibernate, constatem que:

- Via anotació, fa cas del contingut de `@ForeignKey` (nom i definició si s'escau)
- Via marcatge XML, es desentén del marcatge, no en fa cas.

Recordatori: Oracle no admet clàusula `on update` i es generaria un error en cas d'indicar-la.

Tot i així, en el marcatge a aplicar en els exemples i exercicis d'aquesta UF, aplicarem la norma definida per JPA, encara que el proveïdor de la persistència no ho tingui en compte.

Remarquem que, en el món real, l'estructura de la BD ja està creada i el motor de persistència es configura per a que, com a molt, validi la coherència del marcatge amb la definició de les taules.

Continuació pràctica – Relacions ManyToOne - Classe Prestec

20/01/25

JPA sap gestionar tot tipus de relacions entre classes: many2one, one2many, one2one i many2many.

La classe `Prestec` ens obliga a introduir el concepte `ManyToOne`, doncs incorpora les referències:

```
Soci soci
Fitxa fitxa
```

La relació `ManyToOne` | `many-to-one` acostuma a anar acompanyada de `@JoinColumn` | `<join-column>` per definir el nom que ha de tenir la columna i, si cal, nom i definició de la clau forana. Si no s'indica `@JoinColumn` | `<join-column>` JPA assigna a la columna corresponent al camp, un nom que, a diferència dels camps no relacionals, no coincideix amb el nom del camp dins la classe, sinó que munta un nom a partir del nom del camp dins la classe i el nom de la taula apuntada.

Per incidir en el nom (i possible definició) de la clau forana, usarem `@ForeignKey` | `<foreign-key>`, malgrat potser l'eina JPA no en faci cas, com ja s'ha comentat anteriorment.

Projectes solució per marcar classe `Prestec` amb programes de comprovació de funcionament:

- En primer lloc comprovem que la classe `Prestec` ja té els requeriments mínims per poder-la fer persistent (`Serializable`, constructor sense paràmetres, `NO final`, `getter-setter` no final, camps id. immutables)
- Si revisem la classe `Prestec`, observem que la seva lògica no és correcta, doncs en assignar una fitxa al préstec no comprova si la fitxa és prestable i si ja està prestada...
Respecte la comprovació sobre si és prestable, només cal retocar el mètode `setFitxa` incorporant comprovació respecte el contingut del camp `esDeixa` de la fitxa.
Respecte la comprovació sobre si està prestada, un objecte `Prestec` per si sol no ho pot esbrinar amb l'actual definició de les classes; ho solucionem:
 - incorporant camp `deixada` a la classe `Fitxa`, que en crear una fitxa sempre estigui a `false`
 - dins `Fitxa` incorporem corresponents mètodes `getDeixada` i `setDeixada`, i retoquem mètode `toString`
 - dins la classe `Prestec`, en el mètode `setFitxa`, comprovi si la fitxa està disponible
 - dins la classe `Prestec`, en el mètode `setMomentPrestec` marcar la fitxa com a `deixada` i en el mètode `setMomentRetorn`, deixar la fitxa disponible per a nou préstec.
- 250120_1_BibliotecaV1
- 250120_2_BibliotecaV1A
- 250120_3_BibliotecaV1A_Hibernate
- 250120_4_BibliotecaV1X_Hibernate

Respecte els programes de comprovació de funcionament, prèviament eliminar taula `FITXA` de la BD, doncs el procés d'actualització del programa `P01` ha d'incorporar columna `deixada` que és `NOT NULL` i no ho pot fer si la taula ja té files d'anteriors proves.

- `P02` cerca/crea 2 fitxes i 2 socis i intenta prestar les 2 fitxes als 2 socis. Evidentment, només pot prestar una fitxa quan encara no està prestada. A més, calcula el darrer número de préstec existent per assignar números correlatius als nous préstecs que es creïn.
- `P03` retorna tots els préstecs existents pendents de retorn, via consulta JPQL per obtenir-los i posteriorment executar el retorn del préstec via mètode `setMomentRetorn` per cada préstec.
- `P04` demostra que, quan l'`EntityManager` carrega un préstec en memòria, es veu obligat a carregar la corresponent `Fitxa` i el corresponent `Soci` si no els té carregats prèviament. Això es pot veure comprovant les `SELECT` que executa l'eina. I en realitat, com que només vol mostrar número i moments de préstec i de retorn,... no caldria haver carregat el `Soci` i la `Fitxa` corresponents.
En el següent apartat veurem la clàusula `fetch` per aconseguir que `Soci` i `Fitxa` no es carreguin en memòria si el programa no els necessita.

- P05 demostra que, en cas de fer persistent un Prestec sense haver fet persistent prèviament la seva Fitxa i/o Soci, es genera un error.
En el següent apartat veurem la clàusula cascade per aconseguir que en fer persistent un Prestec, els corresponents Soci i Fitxa també passin a ser-ho sense que el programador hagi d'especificar-ho.

Clàusules fetch i cascade en camps relacionals

22/01/25

En tots els camps relacionals és possible configurar 2 actuacions molt importants:

- **fetch:** Permet indicar a l'EntityManager en quin moment carrega en memòria els objectes d'altres classes com a conseqüència de relacions entre les classes (ManyToOne, OneToMany, ManyToMany,...).

És a dir, imaginem que executem la següent instrucció per carregar en memòria el préstec 1000:

```
Prestec p = em.find(Prestec.class, 1000);
```

La classe Prestec conté les referències als corresponents objectes Soci i Fitxa! JPA els carrega també en memòria? Sembla que la resposta hauria de ser afirmativa i... si Soci contingués una referència cap als seus préstecs (List<Prestec> prestecs), implicaria que pel fet de carregar el Soci en memòria, també carregaria els seus préstecs? I cada préstec la seva Fitxa i el seu Soci i cada Soci.... Aquesta problemàtica ja l'hem viscut a la UF3 i allà ja varem veure possibilitats de solució... La bona, que no varem implementar per què és molt-molt-molt costosa, consistia en que algú (en segon pla) s'encarregués de carregar els objectes referenciats en memòria en el moment que es necessitin... Doncs JPA incorpora aquesta solució!!!

La clàusula **fetch** (*anar a cercar*) en els camps relacionals permet decidir l'actuació que es vol:

FetchType.LAZY (actuació gaudula-diferida): Carrega objectes referenciats quan es necessitin

FetchType.EAGER (actuació immediata): Carrega objectes referenciats immediatament

Aquesta clàusula **fetch**, molt interessant en els camps relacionals, és aplicable a tots els atributs (no ñes habitual) i pot ser especialment interessant en atributs "pesats" (BLOB, CLOB,...).

La clàusula **fetch**, per defecte (en tot tipus d'atribut) té el valor **EAGER** excepte en les relacions **OneToMany** i **ManyToMany**, on té el valor **LAZY** (lògic, doncs en aquests casos un objecte pot estar relacionat amb molts-molts objectes)

- **cascade:** Permet indicar a l'EntityManager com actuar sobre els objectes referenciats quan un objecte el fem persistent (**persist**) o quan li apliquem altres accions (**merge**, **refresh**, **remove** i **detach**, encara pendents de conèixer). Actuacions permeses:

| | | |
|----------------------------------|----------------------------------|--|
| <code>CascadeType.PERSIST</code> | <code>CascadeType.REFRESH</code> | <code>CascadeType.DETACH</code> |
| <code>CascadeType.MERGE</code> | <code>CascadeType.REMOVE</code> | <code>CascadeType.ALL</code> (les engloba totes) |

Projectes exemple:

- Retoquem marcatge en classe Prestec fent que els camps relacionals fitxa i soci actuïn amb **fetch LAZY** i amb **cascade PERSIST** i comprovem com canvia el funcionament dels programes P04 i P05 del darrer projecte.
- 250122_1_BibliotecaV1
- 250122_2_BibliotecaV1A
- 250122_3_BibliotecaV1A_Hibernate
- 250122_4_BibliotecaV1X_Hibernate
- Amb el programa P04 es pot observar que l'eina va a cercar els socis i les fitxes a mesura que el programa les necessita.
- Amb el programa P05 es pot observar que en fer persistent el nou préstec, el soci i la fitxa es fan persistents automàticament.

[Vídeo de desenvolupament similar](#)

ALERTES!

- **Hibernate no actua correctament davant FetchType.LAZY si hi ha mètodes setter-getter final.**
Avisa amb warning en crear l'Entity Manager si detecta mètodes setter-getter final.



| |
|--|
| <p>- L'especificació de JAKARTA 3.1, a l'apartat 11.1.30, respecte l'anotació <code>ManyToOne</code> explicita:</p> <p><i>The EAGER strategy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime that the associated entity should be fetched lazily when it is first accessed. The implementation is permitted to eagerly fetch associations for which the LAZY strategy hint has been specified.</i></p> <p>És a dir, el proveïdor de la persistència no està obligat a seguir l'estratègia LAZY, malgrat s'indiqui. És bo usar-la per evitar accessos innecessaris a la BD i esperar que el proveïdor de la persistència la implementi correctament.</p> |
|--|

| Control de profunditat de cerca en relacions <code>OneToOne</code> o <code>ManyToOne</code> | 23/01/25 |
|---|----------|
| <p>Hibernate permet definir la màxima profunditat de cerca en les relacions <code>OneToOne</code> or <code>ManyToOne</code> en les que el funcionament per defecte és EAGER. És una manera d'evitar càrrega inútil d'objectes en memòria per si no s'ha definit <code>fetch</code> a LAZY en alguna relació <code>OneToOne</code> o <code>ManyToOne</code>.</p> <pre><property name="hibernate.max_fetch_depth" value="0" /></pre> <p>Sets a maximum depth for the outer join fetch tree for single-ended associations. A single-ended association is a one-to-one or many-to-one association. A value of 0 disables default outer join fetching</p> <p>En Eclipse???</p> | |

| ATENCIÓ en consultes JPQL on intervinguin vàries classes: JOIN / LEFT JOIN / FETCH | 23/01/25 |
|--|----------|
| <p>A partir d'aquest moment, en el què hem començat a veure com establir relacions entre classes, pot ser que ens interressi efectuar alguna consulta JPQL on es vegin involucrades diverses classes relacionades amb <code>ManyToOne</code>, <code>OneToOne</code>, <code>ManyToOne</code> o <code>OneToMany</code>.</p> <p>Cal tenir present que tots els joins en JPQL són, per defecte, <code>INNER JOIN</code>, és a dir, que no apareixeran els objectes que no tinguin relació i, per evitar-ho, caldrà explicitar <code>LEFT JOIN</code>. I això afecta als joins que hagi de decidir JPA encara que no els haguem detallat nosaltres.</p> <p>Per altra banda, la clàusula <code>JOIN</code> en JPQL pot anar acompanyada de <code>FETCH (JOIN FETCH)</code> per forçar que els objectes relacionats siguin recuperats en la consulta malgrat el marcatge hagués estat definit com LAZY.</p> | |

| Generació automàtica de l'identificador | 23/01/25 |
|--|----------|
| <p>Aplicable a camps identificadors enters. Implica que aquest camp passa a ser governat per JPA.</p> <p>Per tant, cal fer canvis a les classes on es pugui i vulgui aplicar:</p> <ul style="list-style-type: none"> - Retoc del constructor, eliminant el camp identificador. - Eliminar mètode <code>setter</code> corresponent al camp identificador, doncs en cap cas ha de canviar el camp codi. <p>4 estratègies: Tots els projectes exemple que es faciliten tenen un llegeume amb explicacions.</p> <ul style="list-style-type: none"> - AUTO: <ul style="list-style-type: none"> - Cada proveïdor (fabricant) JPA actua com li sembla més convenient en funció del SGBDR. - Algun proveïdor JPA pot, fins i tot, generar el comptador en memòria, de manera que a cada execució es torna a començar. - És només per a proves (desenvolupament). <p>Exemple: <code>Projecte 250123_1_ClauAutomàtica_AUTO_Hibernate</code></p> - IDENTITY: <ul style="list-style-type: none"> - Comptadors implementats amb camps autonumèrics. - Aplicable a SGBD que incorporen camps autonumèrics, com PostgreSQL i MySQL i Oracle 12+ <p>Exemple: <code>Projecte 250123_2_ClauAutomatica_IDENTITY_Hibernate</code></p> - SEQUENCE: <ul style="list-style-type: none"> - Comptadors implementats via seqüències - Aplicable a SGBD que incorporen gestió de SEQUENCE, com Oracle i PostgreSQL. | |

MySQL8 no conté seqüències, però els proveïdors JPA poden simular-les, de manera que també podem utilitzar aquesta estratègia en MySQL:

- Hibernate, per cada taula amb aquesta estratègia, crea una taula amb comptador per simular la seqüència.
- EclipseLink, no crea cap estructura addicional i quan necessita generar el següent valor, cerca el major valor clau existent a la taula i genera clau amb el valor següent.

Exemple: Projecte 250123_3_ClauAutomatica_SEQUENCE_Hibernate

- TABLE:

- Comptadors implementats en taula de comptadors amb estructura similar a la taula de la dreta.
- Una taula de comptadors pot servir per moltes taules
- Aplicable a qualsevol SGBDR
- Alerta amb l'atribut `initialValue` | `initial-value` en Hibernate, doncs el guarda com a `DarrerValor` quan en realitat hauria de guardar-hi `initialValue-1`. En canvi, el funcionament en EclipseLink és correcte.

| IdentificacióDeTaula | DarrerValor |
|----------------------|-------------|
| Taula1 | 23 |
| Taula2 | 45 |
| ... | ... |

Exemple: Projecte 250123_4_ClauAutomatica_TABLE_Hibernate

- UUID (apareix en Jakarta 3.0 – inexistent en JPA 2.2):

Informació sobre UUID: <https://youtu.be/MUK5qZxIWFg>
<https://interactivechaos.com/es/wiki/identificador-unico-universal-uuid>

- Cal que el camp identificador en Java sigui del tipus `UUID` (classe `java.util.UUID`)
- En Oracle, la columna corresponent és del tipus `RAW(16)`.

Exemple: Projecte 250123_5_ClauAutomatica_UUID_Hibernate

Hibernate, per implementar aquestes tècniques, crea taules temporals amb prefix `HT`, que no han de preocupar.

Informació respecte `TableGenerator` i `SequenceGenerator` en EclipseLink

EclipseLink considera que no hi pot haver, a l'aplicació, `TableGenerator` o `SequenceGenerator` amb idèntic nom. Ho considera un error de marcatge. Per tant, si hi ha varies classes amb clau automàtica, per cada classe caldrà definir un `TableGenerator` / `SequenceGenerator` amb nom diferent; això no impedeix, en el cas de `TableGenerator`, que la taula de comptadors sigui la mateixa per a diferents taules.

Informació IMPORTANT per les seqüències en Oracle

El SGBD Oracle inclou a les `SEQUENCE` el concepte `CACHE` per qüestions de rendiment, de manera que quan necessita un nou valor de la seqüència, en "reserva" tants en memòria com indica la propietat `CACHE` i, si consultem el darrer número de la seqüència, veiem el darrer "reservat".

En algunes versions d'Oracle, els números "reservats" i no "assignats" es perden davant un reinici de l'Oracle, fet que provoca "forats" en la generació de codis. Sembla que no passa en Oracle 21.

En PostgreSQL les `SEQUENCE` no tenen aquesta propietat.

En Oracle, es pot desactivar el `CACHE` per a una `SEQUENCE`, però Hibernate la crea amb `CACHE 20` (que és el funcionament per defecte d'Oracle en crear una `SEQUENCE` si no s'especifica res relatiu a la propietat `CACHE`). Aquest problema queda solucionat si la creació de la BD es fa via guió i JPA s'activa amb només validació.

Exercici per dilluns 27/gener - Continuació pràctica

27-01-25

Incorporar clau automàtica a les classes `Soci` i `Prestec`

- Utilitzar l'estratègia `TABLE` que és vàlida per qualsevol SGBDR.
- Comprovar funcionament en Oracle-Hibernate.

Solució:

- Modificar constructor de classe `Soci` eliminant paràmetre `codi`.
- Modificar constructor de classe `Prestec` eliminant paràmetre `numero`.
- Eliminar mètode `setCodi` a `Soci`
- Eliminar mètode `setNumero` a `Prestec`
- Classe `Soci`: Afegir anotació/marcatge XML per assolir generació automàtica de `codi`.
- Classe `Prestec`: Afegir anotació/marcatge XML per assolir generació automàtica de `numero`.

Projectes solució:

Mireu els comentaris que hi ha en `@TableGenerator` relatives a `initialValue` i `allocationSize`

- 250127_1_BibliotecaV1
- 250127_2_BibliotecaV1A
- 250127_3_BibliotecaV1A_Hibernate
- 250127_4_BibliotecaV1X_Hibernate

Quan assigna el valor autonumèric?

En el moment de fer persistent l'objecte (es pot comprovar "debugant" el programa i observant que abans d'executar la instrucció `persist`, la clau té 0 per valor i posteriorment agafa el valor que pertoca.

Com "alterar" la definició de la UP de `persistence.xml` en crear l'`EntityManagerFactory`?

En ocasions pot interessar que, en el moment de carregar la UP, afegir/alterar algunes de les propietats definides a la UP dins el fitxer `persistence.xml`.

Això es pot aconseguir invocant:

```
Persistence.createEntityManagerFactory("nomUP", propietats);
```

on `propietats` és un `Map<String,String>` (per exemple un `HashMap`) que conté les propietats a afegir/alterar (clau) amb el corresponent valor.

En el programa P01 dels projectes de prova anteriors s'ha utilitzat aquesta tècnica per "obligar" a que l'execució de P01 efectués la creació de totes les taules (ja que les dades existents podien interferir en la provatura dels programes), sense modificar `persistence.xml`.

És a dir, dins `persistence.xml` hi ha la propietat:

```
<property name="hibernate.hbm2ddl.auto" value="update" />
```

que ens interessa en els programes P02 i següents, però com que en P01 volíem que actués "create", hem procedit a crear l'`EntityManagerFactory` com segueix:

```
HashMap<String,String> propietats = new HashMap();
propietats.put("hibernate.hbm2ddl.auto", "create");
emf = Persistence.createEntityManagerFactory(up,propietats);
```

Amb aquest codi, aconseguim que la propietat `hibernate.hbm2ddl.auto` definida en el `HashMap`, sobreescriu la definida dins `persistence.xml`. i en executar P01 es recreïn totes les taules.

Aquesta opció s'ha implementat en els programes P01 dels darrers projectes i en els propers.

Herència en Java <=> Especialització disjunta en model ER

29/01/25

Suposem que tenim una classe `Base` (abstracta o no) amb dues classes derivades `Derivada1` i `Derivada2` "disjunes", és a dir, on una instància només pot pertànyer a una única jerarquia. Ho exemplificarem amb classe `Persona` i classes derivades `Alumne` i `Professor` on no es pot ser simultàniament alumne i professor.

Com ha de ser la persistència dels seus objectes en un SGBDR? Com en feríeu el disseny?

Si pensem en el model E-R, segur que dissenyaríeu una entitat `PERSONA` amb una especialització `ALUMNE` i `PROFESSOR`. (parcial/total i disjunta). I, com seria la traducció a un model relacional? Recordareu que hi ha diverses estratègies i JPA permet implementar totes elles.

3 estratègies: **Tots els projectes exemple que es faciliten tenen un llegeume amb explicacions.**

- 1 única taula `PERSONA` per emmagatzemar totes les persones, siguin alumnes o professors. Per tant, tindrà

columnes per les dades de PERSONA i columnes per les dades específiques d'ALUMNE i de PROFESSOR.

Això implica que aquesta taula ha d'admetre molts valors nuls:

- Una fila que emmagatzemi un ALUMNE, tindrà nul a les columnes corresponent a PROFESSOR.
- Una fila que emmagatzemi un PROFESSOR, tindrà a nul les columnes corresponents a ALUMNE.
- Una fila que emmagatzemi una PERSONA que no sigui ni ALUMNE ni PROFESSOR (en cas que es permeti), tindrà a nul les columnes dedicades a les dades específiques d'ALUMNE i a les dades específiques de PROFESSOR.

La taula també ha de contenir una columna que permeti distingir les files de les diverses entitats.

JPA permet aquesta implementació amb l'anomenada estratègia **SINGLE TABLE**.

Projecte amb anotacions: 250129_1A_HerenciaA_SINGLE_TABLE_Hibernate

Projecte amb marcatge XML: 250129_1X_HerenciaX_SINGLE_TABLE_Hibernate

- 1 taula per a cada entitat, totalment independents:
 - Taula ALUMNE pensada per guardar les instàncies que siguin ALUMNE.
 - Taula PROFESSOR pensada per guardar les instàncies que siguin PROFESSOR.
 - Taula PERSONA pensada per guardar les instàncies que siguin PERSONA (ni ALUMNE ni PROFESSOR en cas que es permeti aquest fet)

JPA permet aquesta implementació amb l'anomenada estratègia **TABLE PER CLASS**.

En cas que la classe BASE sigui abstracta, la corresponent taula no té raó d'existir. És a dir, si la classe Persona és abstracta (no hi pot haver objectes Persona), no existirà la seva taula.

Projecte amb anotacions: 250129_2A_HerenciaA_TABLE_PER_CLASS_Hibernate

Projecte amb marcatge XML: 250129_2X_HerenciaX_TABLE_PER_CLASS_Hibernate

- 1 taula amb les dades comunes i taules per les dades específiques de cada classe:
 - Taula PERSONA pensada per guardar les dades comunes
 - Taula ALUMNE pensada per guardar les dades específiques d'alumne, amb FK cap PERSONA
 - Taula PROFESSOR pensada per guardar les dades específiques de professor, amb FK cap PERSONA.

La taula pot contenir o no una columna que permeti distingir les files de cada entitat.

Aquesta opció és la que hem utilitzat més a 1r curs.

JPA permet aquesta implementació amb l'anomenada estratègia **JOINED**.

Projecte amb anotacions: 250129_3A_HerenciaA_JOINED_AmbDiscriminator_Hibernate

Projecte amb marcatge XML: 250129_3X_HerenciaX_JOINED_AmbDiscriminator_Hibernate

Projecte amb anotacions: 250129_4A_HerenciaA_JOINED_SenseDiscriminator_Hibernate

Projecte amb marcatge XML: 250129_4X_HerenciaX_JOINED_SenseDiscriminator_Hibernate

Hibernate, per implementar aquestes tècniques, crea taules temporals amb prefix HT, que no han de preocupar.

ALERTA! JPA pressuposa que la columna identificadora en totes les taules té el mateix nom, que és el que s'hagi donat a la taula base. En cas que es vulgui tenir diferent nom en les taules corresponents a les classes derivades, cal usar `@PrimaryKeyJoinColumn` - `<primary-key-join-column>` en les classes derivades.

És a dir, si la clau primària a F_LLIBRE hagués de ser REF_LLIBRE i/o a F_REVISTA hagués de ser REF_REVISTA enlloc de ser REFERENCIA com a FITXA, hauríem d'usar el marcatge anterior.

A més, el marcatge `@PrimaryKeyJoinColumn` - `<primary-key-join-column>` haurà d'acompanyat de `@ForeignKey` - `<primary-key-foreign-key>` per donar nom a la corresponent clau forana i incorporar la definició de clau forana si es vol forçar on delete, que és habitual en aquests casos, malgrat potser l'eina JPA no en faci cas, com ja s'ha comentat anteriorment.

Exercici - Continuació pràctica – Mètode “remove” per eliminar objecte...

29/01/25

Implementar la persistència de les classes FitxaRevista i FitxaLlibre.

- Tornar a deixar la classe `Fitxa` abstracta, amb mètode `toString` abstracte.
- Utilitzar l'estratègia `JOINED` amb discriminador que és la que estem més acostumats a utilitzar.
- La taula on s'emmagatzemi els llibres s'ha de dir `f_llibre` i la clau primària ha de ser `ref_llibre` i la clau forana respecte `fitxa` s'anomeni `f_llibre_fk_fitxa` amb eliminació en cascada.
- La taula on s'emmagatzemi les revistes s'ha de dir `f_revista` i la PK ha de ser `ref_revista` i la clau forana respecte `fitxa` s'anomeni `f_llibre_fk_fitxa` amb eliminació en cascada..
- Comprovar funcionament en Oracle-Hibernate.

Solució:

- En primer lloc comprovem que les classes `FitxaLlibre` i `FitxaRevista` verifiquen els requeriments mínims per poder-la fer persistent: (constructor sense paràmetres, NO final, getter-setter no final)
- Via anotacions, incorporem anotacions vinculades a estratègia `JOINED` amb discriminador en classe `Fitxa` (que hem fet abstracta) i en classes `FitxaRevista` i `FitxaLlibre`.
- Via marcatge XML, incorporem anotacions vinculades a estratègia `JOINED` amb discriminador per classe `Fitxa` en fitxer `META-INF/fitxa.xml` i en el mateix fitxer incorporem marcatge per classes `FitxaRevista` i `FitxaLlibre`. Podrien estar en fitxers diferents, un per cada classe.
- Taula `fitxa` per enregistrar les dades comunes. Conté columna `TIPUS` discriminadora.
- Taula `f_revista` per enregistrar les dades específiques dels objectes `Revista` amb `ref_revista` com a clau primària i que la clau forana respecte `fitxa` s'anomeni `f_revista_fk_fitxa` amb eliminació en cascada.
- Taula `f_llibre` per enregistrar les dades específiques dels objectes `Llibre` amb `ref_llibre` com a clau primària i que la clau forana respecte `fitxa` s'anomeni `f_llibre_fk_fitxa` amb eliminació en cascada.

Projectes solució:

- 250129_5_BibliotecaV1
- 250129_6_BibliotecaV1A
- 250129_7_BibliotecaV1A_Hibernate
- 250129_8_BibliotecaV1X_Hibernate

Respecte els programes de prova:

- P01: Crea `EntityManager` amb propietat "create" per a que recreï les taules que puguin existir. Comprovar, després d'executar-lo, l'estructura de les 3 taules.
- P02: Crea un objecte `FitxaRevista` i un objecte `FitxaLlibre` i els fa persistents. Comprovar, després d'executar-lo, la informació que hi ha a les 3 taules
- P03: Recupera les fitxes enregistrades en P02. Comprovar, després d'executar-lo, la informació que hi ha a les 3 taules
- P04: Intenta eliminar les fitxes, invocant mètode `remove` per marcar-les com objectes a eliminar. Evidentment, les fitxes que estan prestades no podran ser eliminades quan executi el `delete` en fer `flush` o `commit`. Per veure que s'elimina les que es poden eliminar, invoquem `commit` per a cada fitxa a eliminar enlloc de fer un `commit` global al final.
Observar que, per eliminar una revista/llibre, JAKARTA sap que primer ha d'invocar el `delete` de `f_llibre/f_revista` i posteriorment el `delete` de `fitxa`, i, per tant, no necessita que la clau forana de `f_llibre/f_revista` respecte `fitxa` incorpori on `delete cascade`, però com que pot ser interessant que aquesta FK incorpori on `delete cascade`, ha calgut usar `@ForeignKey - <primary-key-foreign-key>`.

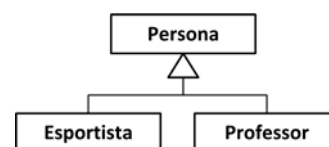
Especialització solapada en model ER – Com es modelitza en POO? Herència en Java?

03/02/25

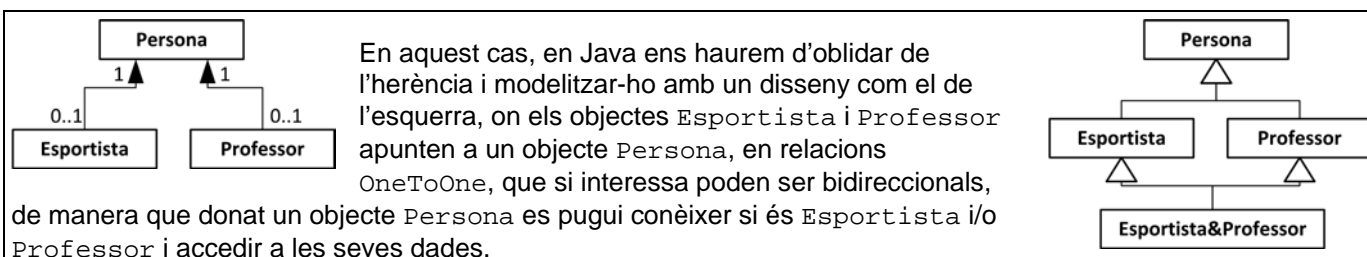
Suposem que hem de dissenyar classes per modelitzar persones que poden ser esportistes i/o professors.

Un disseny de classes com el de la dreta no permet tenir objectes `Persona` que siguin simultàniament `Esportista` i `Professor`.

Hauríem de tenir un disseny com el de la dreta, amb la definició de la classe `Esportista&Professor` que deriva de les classes `Esportista` i `Professor`. Aquest és un cas d'herència múltiple, que Java no suporta.



Llenguatges que permeten herència múltiple: C++, Python, Perl, Eiffel,...



Amb JPA es pot aconseguir persistència en BDR similar a l'estratègia JOINED sense camp discriminador, tenint:

```

PERSONA (#codi, dades_comunes)
ESPORTISTA (#codi, dades_específiques_esportista) on {dni} referencia PERSONA
PROFESSOR (#codi, dades_específiques_professor) on {dni} referencia PERSONA

```

Per aconseguir-ho, a les classes Esportista i Professor cal tenir en compte:

- Camp `codi` del mateix tipus que el camp `codi` de Persona marcat amb `@id`.
- Camp apuntant a Persona, marcat amb [@OneToOne/one-to-one](#) i amb `@MapsId/maps_id` per forçar que el camp `@Id` de la classe s'empleni automàticament amb el valor del camp `@Id` de la classe Persona. Incorporarem `@JoinColumn` per assignar el nom que calgui a la col. d'enllaç a taula PERSONA.
- Molt aconsellable que la OneToOne d'Esportista/Professor a Persona sigui `Cascade.PERSIST`.
- En cas que des de Persona es vulgui conèixer si és Esportista i/o Professor, caldrà definir dins Persona referències no obligatòries a Esportista i/o Professor marcades amb OneToOne inversa.
- Molt aconsellable que la OneToOne de Persona a Esportista/Professor sigui `Cascade.REMOVE`.

Alerta!

Ens podem estalviar dins Esportista/Professor el camp `codi` del mateix tipus que el camp `codi` de Persona marcat amb `@id` i usar el marcatge següent?

```

@Id
@OneToOne (...)
@JoinColumn(name="dni", nullable=false,
            foreignKey=@ForeignKey(name="professor_fk_dni"))
private Persona persona;

```

No! Hibernate (EclipseLink ???) no es queixa però a l'hora d'enregistrar un esportista/professor a la BD, no executa `INSERT INTO PERSONA` i només fa `INSERT INTO ESPORTISTA/PROFESSOR` generant un error d'integritat referencial.

Projectes exemple: 250203_1_EspecialitzacioSolapadaA_Hibernate
250203_2_EspecialitzacioSolapadaA_Hibernate
Veure llegime dins projectes amb explicació de necessitat de LEFT JOIN. (P04)

Herència amb classe base no persistent

05/02/25

Cal marcar la classe amb [@MappedSuperclass/mapped-superclass](#) enlloc de `@Entity/entity`. Les classes derivades, si són persistents, cal marcar-les amb `@Entity/entity`. Per cada classe derivada persistent es crearà una taula que contindrà les columnes corresponents a camps de la classe base i les columnes de la pròpia classe.

Aquesta opció pot ser interessant quan la classe base és abstracta, però també te sentit usar-la encara que no sigui abstracta i els seus objectes no hagin de ser mai persistents.

Alerta: No existeixen entitats de la classe base i no es pot cercar (`find-JPQL`) per aquesta classe.

Exercici: Donat que la classe Fitxa és abstracta, fem una versió fent-la `@MappedSuperclass`.

Comentaris:

- No es pot mapar la classe Prestec, doncs el camp `fitxa` és referència a Fitxa i això obliga a que Fitxa sigui `@Entity` o `@Embeddable` (que està per veure) però no pot ser `@MappedSuperClass`.
- Dins `persistence.xml` cal comentar la càrrega de la classe Prestec.
- El programa P01 no elimina la taula FITXA existent d'anteriors programes. Cal fer drop manual.
- El programa P02 no pot fer un `find(Fitxa.class...` Cal saber si es va a cercar un FitxaRevista o

un FitxaLlibre.

Projectes solució:

- 250205_1_BibliotecaV1
- 250205_2_BibliotecaV1A
- 250205_3_BibliotecaV1A_Hibernate
- 250205_4_BibliotecaV1X_Hibernate

Objectes incrustats (classes @Embeddable/embeddable)

05/02/25

Fins ara, els camps de les nostres classes persistents han estat de tipus primitius o de classes facilitades per Java o de camps relacionals (ManyToOne) cap a altres classes persistents. Però... i si tenim una classe no persistent A que utilitzem en una classe persistent P, com per exemple:

- Classe Adreça que utilitzem de manera unívoca (només una instància) a diferents classes (Client, Proveïdor, Comanda,...)?

L'existència de la classe Adreça està justificada per garantir uniformitat (mateixos camps i mateixa funcionalitat) en totes les classes on s'utilitzi.

- Classe CorreuElectronic, amb mateixa situació que la classe Adreça anterior.

Certament, a nivell de BD, podríem tenir una taula ADREÇA i una taula CORREU-E, amb PK i que s'hi fes referència des de les taules CLIENT/PROVEÏDOR/COMANDA/... però quan aquestes entitats (client/proveïdor/comanda) només tenen UN objecte Adreça i/o CorreuElectronic, és habitual que aquest estigui incrustat dins la taula CLIENT/PROVEÏDOR/COMANDA/...

Cal poder incrustar objectes. Provem-ho a la classe Persona:

```
class Persona {
    ...
    CorreuElectronic correuElectronic;
};
```

La classe CorreuElectronic cal marcar-la com @Embeddable/embeddable (no com a @Entity/entity que s'utilitza per a les classes persistents).

Dins la classe Persona cal marcar l'objecte incrustat amb @Embedded/embedded.

Les columnes corresponents a CorreuElectronic, dins la taula Persona, hereten nom i atributs segons definició dins la classe CorreuElectronic i si es volen alterar (doncs CorreuElectronic es pot utilitzar en moltes classes) cal utilitzar @AttributeOverrides/attribute-override.

Projecte amb anotacions: 250205_A_ObjecteIncrustatA_Hibernate
Projecte amb marcatge XML: 250205_A_ObjecteIncrustatX_Hibernate

Els projectes contenen un llegime amb informació.

Col·leccions d'objectes bàsics (classes facilitades per Java)

05/02/25

Ho posem en pràctica incorporant a la classe Persona la possibilitat d'introduir els telèfons (objectes String)

```
class Persona {
    ...
    List<String> telefon;
};
```

La implementació lògica consisteix en generar una taula que "pengi" de la taula PERSONA que tingui els telèfons.

Per aconseguir-ho tenim les marques:

- @ElementCollection/element-collection: Per indicar que el camp és una col·lecció
- @CollectionTable/collection-table: Per definir com ha de ser la taula que recull els elements

Permet especificar:

- name: el nom de la taula
 - joinColumns/join-column: el nom de la columna que fa d'enllaç i el nom i definició de la FK (**compte!**)
 - uniqueConstraints/unique-constraint: les restriccions d'unicitat que pugui interessar (no es pot definir PK).
- @Column/column: Per indicar el nom que la columna que conté el valor de la col·lecció ha de tenir dins la nova taula.

Projecte amb anotacions: 250205_B_ColleccioObjectesBasicsA_Hibernate

Projecte amb marcatge XML: 250205_B_ColleccioObjectesBasicsX_Hibernate

Els projectes contenen un llegiume amb informació.

El programa P05 permet afegir un telèfon a la persona i posteriorment mostra la llista de telèfons de la persona i s'observa que sempre afegim pel final de la llista de telèfons. Però en recuperar els telèfons de la BD, per exemple via programa P03, observem que els mostra en un ordre que no és el que tenia la llista.

Per preservar l'ordre dels elements a la llista, cal utilitzar @OrderColumn/order-column com en:

Projecte amb anotacions: 250205_C_ColleccioObjectesBasicsAmbOrdreA_Hibernate

Projecte amb marcatge XML: 250205_C_ColleccioObjectesBasicsAmbOrdreX_Hibernate

Si observeu la taula PERSONA_TELEFONS, la solució ha passat per incorporar una nova columna, que en el projecte hem anomenat ORDRE, de tipus numeric(2) i que conjuntament amb DNI forma la clau primària.

Si s'utilitza @OrderColumn/order-column sense detallar nom ni tipus, el proveïdor JPA decideix.

Col·leccions d'objectes incrustats (classes @Embeddable/embeddable)

05/02/25

Ho posem en pràctica incorporant a la classe Persona la possibilitat d'introduir els seus correus electrònics, però enlloc de fer-ho amb un simple String, utilitzem la classe CorreuElectronic:

```
class Persona {
    ...
    List<CorreuElectronic> correusElectronics;
};
```

En aquest cas cal combinar el marcatge que s'ha vist en els projectes anteriors:

- ObjecteIncrustat
- ColleccioObjectesBàsics

Projectes exemple (que incorporen també la marca @OrderColumn/order-column per mantenir/traslladar l'ordre dels objectes de la col·lecció a la classe dins la taula):

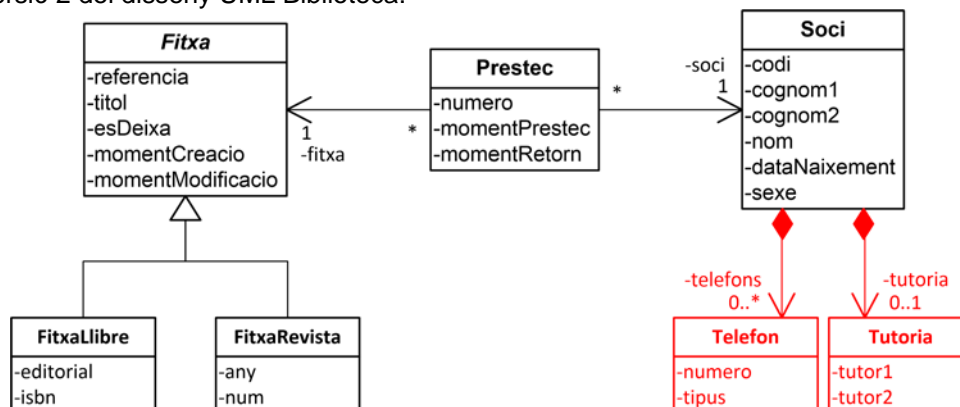
Projecte amb anotacions: 250205_D_ColleccioObjectesIncrustatsA_Hibernate

Projecte amb marcatge XML: 250205_D_ColleccioObjectesIncrustatsX_Hibernate

Exercici - Continuació pràctica

06/02/25

Considerar la versió 2 del disseny UML Biblioteca:





que incorpora les classes (en vermell):

```
class Telefon {
    String numero; // Obligatori i amb contingut
    char tipus; // Valors vàlids: (M)obil - (F)ix
};

class Tutoria {
    String tutor1; // Obligatori i amb contingut
    String tutor2; // No obligatori o amb contingut
};
```

Cal interpretar:

- Un soci pot no tenir tutoria o tenir-ne, i en aquest segon cas, obligatori el tutor1.
- Un soci pot no tenir telèfon o tenir-ne 1 o més.

Exercici:

- Desenvolupar BibliotecaV2 per adequar-se al nou model.
- Efectuar el marcatge via anotacions i via XML. Es demana:
 - Els telèfons de socis quedin enregistrats en taula de nom SOCI_TELEFONS, amb disseny: SOCI_TELEFONS (#soci, #numero, tipus) on {soci} REFERENCIA SOCI
 - La tutoria de socis ha de quedar enregistrada en columnes tutor1 i tutor2 dins taula SOCI.
- Comprovar funcionament en Oracle-Hibernate

Solució:

- Disseny de les classes Telefon i Tutoria.
Recordar Serializable i constructor sense paràmetre i cap getter-setter final.
- Incorporar a la classe Soci el camp tutoria amb els corresponents set i get.
- Incorporar a la classe Soci el camp telèfons (List<Telefon>) i mètodes adequats: addTelefon, removeTelefon i iteTelefons.

Projecte 250206_1_BibliotecaV2

- Anotacions en classe Telefon: @Embeddable i anotacions en camps.
- Anotacions en classe Tutoria: @Embeddable i anotacions en camps.
Malgrat el camp tutor1 és obligatori per Java en un objecte Tutoria, no podem posar nullable=false ni optional=false per que generaria la columna a la taula com a NOT NULL i obligaria a que un soci sempre tingués un tutor1, cosa que no és obligatòria.
- Anotacions en classe Soci, en camps telefons i tutoria per aconseguir requeriments.

Projecte 250206_2_BibliotecaV2A

A partir d'aquest projecte, incorporem una classe Utils amb mètode mostrarInstruccionsSQL() que usarem en tots els programes per permetre decidir si es vol visualitzar les instruccions SQL, de manera que no haguem d'anar modificant la propietat dins persistence.xml.

- Proves de funcionament via anotacions:
 - P01: Crea EntityManager amb propietat "create" per a que recreï les taules que puguin existir. Comprovar, després d'executar-lo, l'estructura de taula SOCI i SOCI_TELEFONS.
 - P02: Crea objectes Soci amb telèfons i diversos tipus de tutoria. Comprovar, després d'executar-lo, la informació que hi ha a les taules
 - P03: Recupera els socis de la taula i els mostra.

Projecte 250206_3_BibliotecaV2A_Hibernate

- Marcatge XML de classe Telefon en fitxer META-INF/telefon.xml
- Marcatge XML de classe Tutoria en fitxer META-INF/tutoria.xml
- Ampliem marcatge XML de classe Soci (camps telefons i tutoria)
- Afegir mapping-file adequats en fitxer persistence.xml

Projecte 250206_4_BibliotecaV2X_Hibernate

Claus compostes constituïdes per camps bàsics

10/02/25

Com actuar davant una clau composta? Ho exemplifiquem amb la classe Color amb implementació:


```
class Color {  
    private int red,blue,green;  
    private String description;  
};
```

Un color queda identificat amb els tres valors red-blue-green. Com implementar-ho?

- **Possibilitat 1 – Insuficient – Marcant amb @Id cada camp clau sense fer res més...**

Projecte amb anotacions: 250210_1A_ClauCompostaViaMultiplesIdA_Hibernate
Projecte amb marcatge XML: 250210_1X_ClauCompostaViaMultiplesIdX_Hibernate

- P01 - La taula es crea amb PK composta. Ok!
- P02 - La persistència d'objectes (inserció de files) s'efectua correctament. Ok!
- P03 - La consulta (via JPQL) funciona correctament. Ok!

Llavors, per què és insuficient? Doncs per que NO permet el mètode `EntityManager.find()` per cercar objectes, doncs aquest mètode només permet indicar un valor per identificar l'objecte a cercar i, si la clau és composta, cal més d'un identificador.

- **Possibilitat 2 – Correcta – Marcant amb @Id cada camp amb classe auxiliar @IdClass** <= + FÀCIL!!!

Es manté el marcatge de cada camp clau amb `@Id/id` i es dissenya una classe de suport que:

- No ha de tenir cap marca (anotació/marcatge XML)
 - Ha d'implementar la interfície `serializable`.
 - Ha de contenir mateixos camps que els camps clau de la classe original
 - Ha de contenir constructor sense paràmetres (`public` o `protected`)
 - Ha de contenir constructor amb tots els paràmetres
- S'utilitzarà per crear un objecte multiclau a utilitzar en el mètode `EntityManager.find()`

La classe original ha de contenir la marca (`@IdClass/id-class`):

- Via anotacions: `@IdClass(nomClassId.class)` abans de la definició de la classe
- Via marcatge XML: `<id-class class="nompaket.nomClassId"/>` dins element `<entity>`

La classe de suport s'acostuma a anomenar igual que la classe original amb prefix `Pk` o `Id`

Projecte amb anotacions: 250210_2A_ClauCompostaViaIdClassA_Hibernate
Projecte amb marcatge XML: 250210_2X_ClauCompostaViaIdClassX_Hibernate

- P01/P02/P03 – Igual que en possibilitat anterior.
- P04 – Mostra que es pot utilitzar el mètode `EntityManager.find()`.

- **Possibilitat 3 – Correcta – Classe auxiliar EMBEDDABLE que contingui els camps clau**

Es dissenya una classe de suport que:

- Declarada com `@Embeddable/embeddable`
 - Ha d'implementar la interfície `serializable`.
 - Ha de contenir mateixos camps que els camps clau de la classe original:
 - o No ha de contenir marques `@Id/id` (les classes incrustades no en poden tenir)
 - o Cada camp ha de contenir les marques `basic, column...` que corresponguin
 - Ha de contenir constructor sense paràmetres (`public` o `protected`)
 - Ha de contenir constructor amb tots els paràmetres
- S'utilitzarà per crear un objecte multiclau a utilitzar en el mètode `EntityManager.find()`

La classe original:

- No ha de contenir els camps clau
- Ha de contenir un objecte de la classe incrustada, amb marca `@EmbeddedId/embedded-id`

La classe de suport s'acostuma a anomenar igual que la classe original amb prefix `Pk` o `Id`

Projecte amb anotacions: 250210_3A_ClauCompostaViaEmbeddedIdA_Hibernate

Projecte amb marcatge XML: 250210_3X_ClauCompostaViaEmbeddedIdX_Hibernate

- P01/P02/P03 – Igual que en possibilitat anterior.
- P04 – Mostra que es pot utilitzar el mètode `EntityManager.find()`.

Claus compostes constituïdes per algun(s) camp(s) `ManyToOne` o `OneToOne`

12/02/25

Com actuar davant una clau composta que conté algun(s) camp(s) `ManyToOne`? Ho exemplificarem amb les classes `Pais` i `Provincia` amb implementació:

```
public class Pais {
    private String codi;        // Obligatori - ISO 3166-1 alfa-2, codis de dues lletres.
    private String nom;        // Obligatori - Llargada màxima 60
};

public class Provincia implements Serializable {
    private Pais pais;          // Obligatori
    private String codi;        // Obligatori - ISO 3166-2 codis de fins a 3 lletres
    private String nom;        // Obligatori amb contingut de 60 caràcters màxim
};
```

A nivell de la BD, la taula província ha de tenir per PK: `codi_pais`, `codi_prov`. Com implementar-ho?

Evidentment el camp `pais` de la classe `Provincia` cal marcar-lo com a `ManyToOne` i per la clau composta podem usar qualsevol de les dues opcions: `IdClass` o `EmbeddedId`

Projectes exemples (contenen arxiu `llegiume` amb informació important):

- Amb `IdClass`:
Projecte 250212_1A_ClauCompostaAmbMany2OneViaIdClassA-V01_Hibernate
- Amb `EmbeddedId`:
Projecte 250212_2A_ClauCompostaAmbMany2OneViaEmbeddedIdA-V01_Hibernate

En ambdós projectes, a més de les observacions de l'arxiu `llegiume`, **observeu la consulta JQPL de P03** que:

- Conté `LEFT JOIN`
- No recupera objectes sencers, sinó columnes i, llavors, el mètode `Query.getResultList()` no retorna una llista d'objectes, sinó una llista d'array d'objectes (`List<Object []>`) i es responsabilitat del programador saber el cast que ha d'efectuar a cada columna.

L'arxiu `llegiume` fa èmfasi amb el problema a l'hora d'utilitzar el mètode `EntityManager.find()`.

Per altra banda, `EclipseLink` NO permet la implementació anterior.

- Amb `IdClass`:
Projecte 250212_3A_ClauCompostaAmbMany2OneViaIdClassA-V01_EclipseLink
En executar P01:

The derived composite primary key attribute [pais] of type [info.infomila.jpa.Pais] from [info.infomila.jpa.ProvinciaId] should be of the same type as its parent id field from [info.infomila.jpa.Pais]. That is, it should be of type [java.lang.String].

- Amb `EmbeddedId`:
Projecte 250212_4A_ClauCompostaAmbMany2OneViaEmbeddedIdA-V01_EclipseLink

En executar P01:

The mapping [pais] from the embedded ID class [class info.infomila.jpa.ProvinciaId] is an invalid mapping for this class. An embeddable class that is used with an embedded ID specification (attribute [provinciaId] from the source [class info.infomila.jpa.Provincia]) can only contain basic mappings. Either remove the non basic mapping or change the embedded ID specification on the source to be embedded.

Versions que permeten utilització "fàcil" de mètode `find` i vàlides per `Hibernate` i `EclipseLink`:

- Amb `@idClass`: Retocs a efectuar a la classe que conté la clau composta **<= + FÀCIL!!!**
 - Per cada camp `ManyToOne` o `OneToOne`, afegir un camp bàsic que coincideixi amb la columna que a la



BD correspon amb el camp ManyToOne o OneToOne i marcar-lo com a @Id/id

- Mantenir el camp ManyToOne o OneToOne, però treure-li la marca @Id/id i incorporar dins @JoinColumn que JPA no l'ha d'utilitzar per en insercions i en modificacions: insertable=false i updatable=false.
- Els mètodes que emplenin el camp ManyToOne, també han d'emplenar el camp bàsic incorporat.
- Ja que hem de crear un nou camp bàsic que JPA només utilitza per la gestió de PK, podem batejar-lo amb un "nom adequat" per a que la PK creï la PK amb l'ordre adequat en els seus camps (veure comentari al respecte dins l'legiume dels projectes anteriors).
- Dins la classe que fa de IdClass, incorporar les columnes que estan marcades amb @Id/id a la classe per la que fa de clau composta. No ha de contenir el camp ManyToOne o OneToOne.
- Interessa que disposi de dos constructors (ho exemplifiquem amb ProvinciaId):
public ProvinciaId(String codiPais , String codiProv)
public ProvinciaId(Pais pais, String codiPais)

Projecte 250212_5A_ClauCompostaAmbMany2OneViaIdClassA-V02_Hibernate
Projecte 250212_5A_ClauCompostaAmbMany2OneViaIdClassA-V02_EclipseLink

En EclipseLink, en crear les taules, dona un error estrany... però les crea.

• **Amb @EmbeddedId:** Retocs a efectuar a la classe que conté la clau composta

- El camp ManyToOne o OneToOne que forma part de la clau composta, mantenir-lo a la classe original (no a la classe incrustada) amb marcatge ManyToOne o OneToOne però sense @Id/id i incorporar dins @JoinColumn que JPA no l'ha d'utilitzar per en insercions i en modificacions: insertable=false i updatable=false.
- A la classe incrustada substituir camp ManyToOne o OneToOne per camp bàsic que coincideixi amb la columna que a la BD correspon amb el camp ManyToOne o OneToOne. Podem batejar els camps bàsics de la classe incrustada de forma "adequada" per què la PK es construeixi en l'ordre "adequat".
- Interessa que disposi de dos constructors (ho exemplifiquem amb ProvinciaId):
public ProvinciaId(String codiPais , String codiProv)
public ProvinciaId(Pais pais, String codiPais)

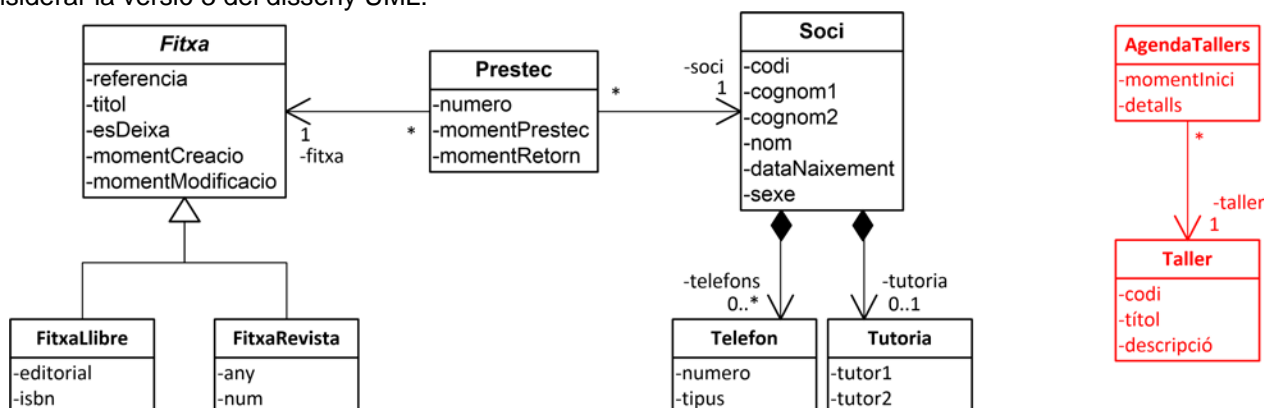
Projecte 250212_6A_ClauCompostaAmbMany2OneViaEmbeddedIdA-V02_Hibernate
Projecte 250212_6A_ClauCompostaAmbMany2OneViaEmbeddedIdA-V02_EclipseLink

En EclipseLink, en crear les taules, dona un error estrany... però les crea.

Exercici - Continuació pràctica

12/02/25

Considerar la versió 3 del disseny UML:



que incorpora les classes (en vermell):

```

public class Taller implements Serializable {
    private String codi;           // Obligatori i de llargada 4
    private String titol;         // Obligatori i de llargada màxima 30
    private String descripcio;    // No obligatori o amb contingut
}
  
```

```

};

public class AgendaTallers implements Serializable {
    private Taller taller;           // Obligatori
    private Calendar momentInici;    // Obligatori
    private String detalls;          // No obligatori o amb contingut
};

```

per gestionar les activitats (Taller) que organitza la biblioteca al llarg del temps (AgendaTaller).

Exercici:

- Dissenyar les noves classes Taller i AgendaTaller, segons disseny.
- Els camps `descripcio` i `detalls` és adequat que a nivell de BD siguin de tipus CLOB o equivalent. Per aconseguir-ho, cal usar la marca `@Lob/lob`.
En Oracle, JPA-Hibernate crea la columna de tipus CLOB.
En MySQL/MariaBD, JPA-Hibernate crea la columna de tipus longtext
En PostgreSQL, JPA-Hibernate hauria de crear la columna de tipus text, però la crea oid. ¿?¿?¿?
- Els objectes Taller han de quedar enregistrats en taula TALLER amb codi com a PK.
- Els objectes AgendaTaller en taula AGENDA_TALLERS on (codi_taller, moment_inici) sigui PK.
Per aconseguir la clau composta, usar `@IdClass/id-class` o `@EmbeddedId/embedded-id`.
- Comprovar funcionament en Oracle-Hibernate

Solució utilitzant `@IdClass` per la clau composta de la classe AgendaTallers:

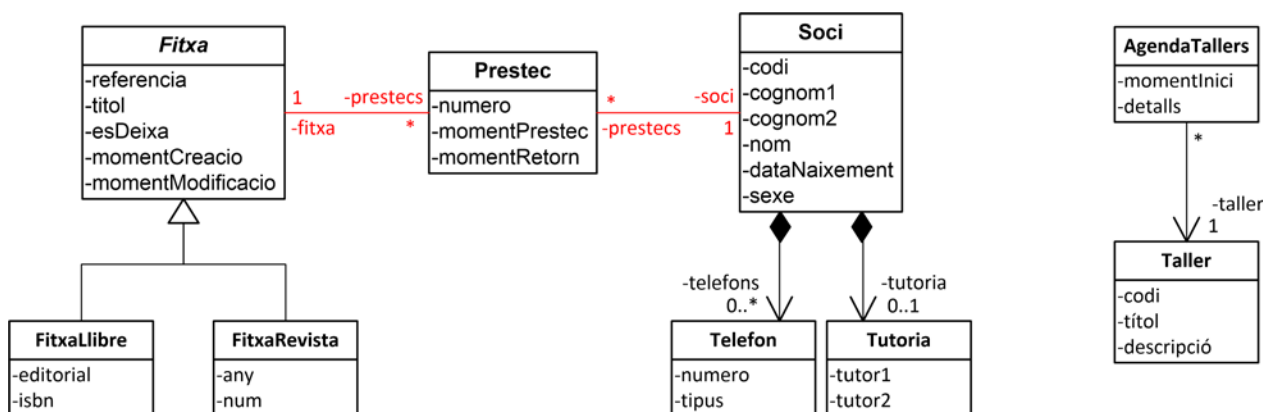
- Projecte 250212_7_BibliotecaV3
- Projecte 250212_8_BibliotecaV3A
- Projecte 250212_9_BibliotecaV3A_Hibernate
- Projecte 250212_A_BibliotecaV3X_Hibernate

El programa P03_MostrarDades mostra el funcionament d'una consulta JPQL amb JOIN (left join) entre classes i, com recollir les columnes de la SELECT on no s'ha demanat un objecte sinó un o varis camps d'objecte.

Continuació pràctica – Relacions OneToMany – Accés a Prestec des de Fitxa i Soci

13/02/25

Considerar la versió 4 del disseny UML:



que estableix com a bidireccionals les relacions entre Prestec i Fitxa/Soci (en vermell).

La versió 1 del disseny UML contemplava les relacions unidireccionals Many2One:

- fitxa de Prestec a Fitxa, per poder saber la Fitxa a la que es refereix el Prestec.
- soci de Prestec a Soci, per poder saber el Soci que efectua el Prestec.

La versió 4 suposa una ampliació de la funcionalitat, doncs ara des d'un objecte Fitxa i/o Soci es podrà navegar als seus objectes Prestec. Això implica implementar les relacions:

- prestecs de Fitxa a Prestec, per poder navegar des d'un objecte Fitxa als seus objectes Prestec (els préstecs efectuats de la fitxa).

- préstecs de Soci a Prestec, per poder navegar des d'un objecte Soci als seus objectes Prestec (els préstecs efectuats pel soci).

Això implica afegir una col·lecció de Prestec dins classes Soci i Fitxa i marcar-la com a OneToMany. La implementació es pot dur a terme amb qualsevol tipus de col·lecció. La solució que implementem nosaltres és:

```
List<Prestec> prestecs = new ArrayList()
```

Aquest nou camp a les classes Soci i Fitxa cal marcar-lo com OneToMany (veure el codi en els projectes).

L'anotació/marca OneToMany acostuma a acompanyar una anotació/marca inversa ManyToOne. (com en Odoo – M10/UF2) i en aquest cas, en efectuar el marcatge OneToMany indicarem la seva inversa ManyToOne amb la clàusula mappedBy. Però NO és obligatori, tot i que nosaltres la utilitzarem sempre amb la inversa ManyToOne.

Explicació detallada: [The best way to map a @OneToMany relationship with JPA and Hibernate](#)

Fonamental: JPA no gestiona la sincronització de les relacions OneToMany i ManyToOne entre classes: les classes han de facilitar la **sincronització** que correspongui (segons el cas). **Potser ho heu vist a M05-UF3**

En el cas que ens ocupa, fem els retocs següents (veure el codi en els projectes):

- Classe Prestec: Retoquem mètodes setSoci i setFitxa, de manera que en assignar un Soci/Fitxa, s'afegeixi el préstec a la corresponent llista prestecs del Soci/Fitxa.
- Classes Soci-Fitxa: Afegim mètodes:
 - itePrestecs per disposar d'un iterator que permeti accedir als préstecs de la llista.
 - addPrestec per afegir un prestec a la llista i serà invocat en els mètodes setSoci i setFitxa. En aquest cas, aquest mètode no és públic (no s'ha de poder invocar des d'una aplicació) degut a:
 - o Els conceptes Soci i Fitxa d'un Prestec són immutables (per criteri inicial – podrien no ser-ho). És a dir, no es permet canviar el Soci ni la Fitxa d'un Prestec.
 - o Suposem que es pogués utilitzar. Ho exemplifiquem a la classe Soci (igual a la classe Fitxa). Suposant que tenim un Soci s i un Prestec p (el qual, si està creat, ja té el seu Soci x, que no té per què ser s. L'execució s.addPrestec(p); implicaria canvi de soci, cosa que no es permet.

És a dir, els mètodes de sincronització cal implementar-los tenint en compte que el soci i la fitxa d'un préstec són obligatoris i immutables. I en aquest cas no te sentit incorporar a les classes Soci-Fitxa, un mètode remove per eliminar un prestec de la llista, doncs aquest no pot quedar sense Soci-Fitxa ni canviar de soci-fitxa.

No sempre serà així. En un model d'empresa, on els clients poden tenir assignat un empleat que els atengui i aquest empleat pot canviar, en cas que la relació entre Client i Empleat sigui bidireccional, tindríem:

- Classe Client, que contindria un camp Empleat. El mètode setEmpleat ha de:
 - Permetre deixar el client sense empleat i canviar l'empleat assignat al client.
 - En cas d'assignar un empleat, afegir el client a la llista de clients de l'empleat.
 - Si el client tenia assignat un empleat i s'ha canviat, treure el client a la llista de clients de l'empleat antic.
- Classe Empleat, que contindria un camp List<Client> o similar. Tindríem els mètodes:
 - iteClients per disposar d'un iterator que permeti accedir als clients de la llista
 - addClient per afegir un client a la llista (mètode public). Aquest mètode ha de controlar:
 - Afegir un client que no té empleat assignat => El client ha de quedar amb empleat assignat.
 - Afegir un client que té assignat un altre empleat => El client ha de quedar amb el nou empleat i ha de desaparèixer de la llista de clients de l'empleat antic, si tenia empleat assignat.
 - removeClient per eliminar un client de la llista (mètode public). Aquest mètode té sentit d'existir doncs l'eliminació del client de la llista ha de deixar el client sense empleat assignat (cosa que és factible)

Per aconseguir tot això, el mètode setEmpleat invoca adequadament els mètodes addClient i removeClient i aquests mètodes invoquen adequadament setEmpleat, sense entrar en un bucle infinit.

En el marcatge OneToMany també podem/cal incorporar de manera adequada la clàusula cascade.

Projectes solució:

- 250213_1_BibliotecaV4
- 250213_2_BibliotecaV4A

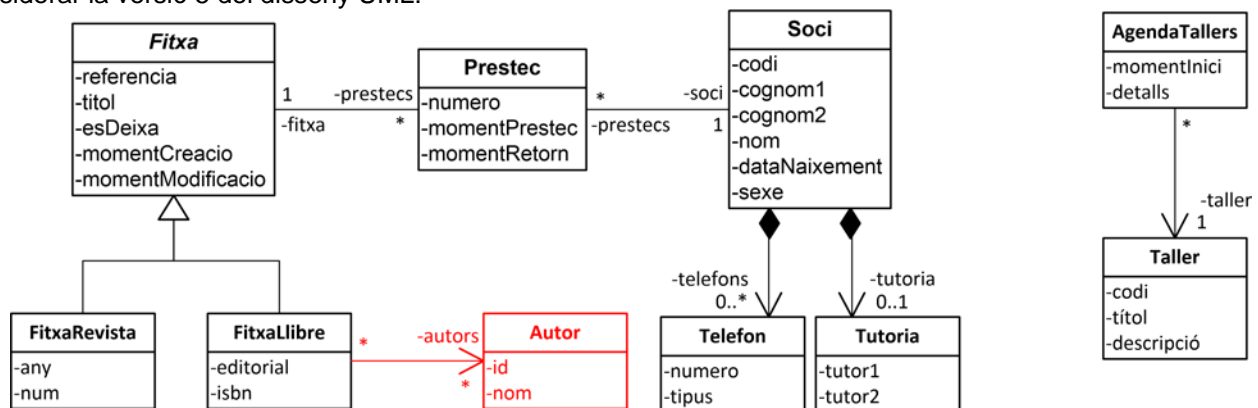


- 250213_3_BibliotecaV4A_Hibernate
- 250213_4_BibliotecaV4X_Hibernate

Continuació pràctica – Relacions ManyToMany unidireccional (Autoria de Llibre a Autor)

17/02/25

Considerar la versió 5 del disseny UML:



que incorpora (en vermell) la nova classe Autor (camps String; evidentment podria tenir molts més camps) i la relació autoria entre FitxaLlibre i Autor. La relació autoria és ManyToMany doncs un llibre pot ser escrit per varis autors i un autor pot haver escrit varis llibres.

A nivell de model relacional, de tots és conegut que aquesta relació entre les taules f_llibre i autor es formalitza amb una taula:

AUTORIA (#llibre, #autor) on llibre REFERENCIA LLIBRE i autor REFERENCIA AUTOR

En aquesta versió, ens interessa implementar la relació autoria de forma unidireccional de FitxaLlibre a Autor, és a dir, des d'un llibre hem de poder navegar/accedir als autors però des d'un autor no podem navegar/accedir als seus llibres.

Per aconseguir-ho, cal incorporar dins FitxaLlibre la col·lecció dels seus autors (incorporant mètodes add, remove, iterator, ...que convingui) i utilitzar les marques/anotacions:

- ManyToMany, per indicar el tipus de relació
- JoinTable, per introduir tota la informació relativa a la taula autoria (nom, PK i FKs)

Projectes:

- 250217_1_BibliotecaV5
- 250217_2_BibliotecaV5A
- 250217_3_BibliotecaV5A_Hibernate
- 250217_4_BibliotecaV5X_Hibernate

Respecte els projectes:

- Observar que s'ha afegit la classe Autor i s'ha completat la classe FitxaLlibre afegint la llista d'autors i els mètodes addAutor, removeAutor i iteAutor.
- Observar l'anotació ManyToMany a FitxaLlibre en projecte BibliotecaV5A i la corresponent marca many-to-many a FitxaLlibre en fitxer.xml de projecte BibliotecaV5X_Hibernate.
- La @JoinTable | join-table incorporada als projectes podria ser:

```

@JoinTable(
    name = "autoria", // nom de la taula
    joinColumns = @JoinColumn(name = "llibre", referencedColumnName = "ref_llibre",
        foreignKey = @ForeignKey(name = "autoria_fk_f_llibre")), // def. columna "llibre"
    inverseJoinColumns = @JoinColumn(name = "autor", referencedColumnName = "id",
        foreignKey = @ForeignKey(name = "autoria_fk_autor")), // def. columna "autor"
    uniqueConstraints = @UniqueConstraint(columnNames = {"llibre", "autor"},
        name = "autoria_pk_llibre_autor") // def PK
)
  
```

Amb aquesta definició, la clau forana no queda amb el on delete cascade, que en alguns casos pot ser

adequat. En el nostre cas, és adequat que a nivell d'SQL:

- L'eliminació d'un llibre a la BD impliqui l'eliminació en cascada de les seves aparicions dins la taula AUTORIA i, per tant, la FK de la taula AUTORIA cap LLIBRE cal que tingui on delete cascade. Per aconseguir-ho cal incorporar dins la corresponent @ForeignKey l'atribut foreignKeyDefinition, amb la definició de la clau forana, tal i com està en el projecte.
- L'eliminació d'un autor no ha de ser possible si te referències des de la taula AUTORIA.

Algunes versions d'Hibernate i EclipseLink "passen" de la definició de l'atribut foreignKey dins @JoinColumn. En canvi, usant la següent definició, que també és vàlida segons l'estàndard JPA, les claus foranes queden creades ok:

```
@JoinTable(
    name = "autoria", // nom de la taula
    joinColumns = @JoinColumn(name = "llibre", referencedColumnName = "ref_llibre"),
    foreignKey = @ForeignKey(name = "autoria_fk_f_llibre", foreignKeyDefinition="..."),
    inverseJoinColumns = @JoinColumn(name = "autor", referencedColumnName = "id"),
    inverseForeignKey = @ForeignKey(name = "autoria_fk_autor", foreignKeyDefinition="..."),
    uniqueConstraints = @UniqueConstraint(columnNames = {"llibre", "autor"}),
    name = "autoria_pk_llibre_autor") // def PK
)
```

En marcatge XML podem observar que Hibernate no fa cap cas de ForeignKey (la crea "al seu aire")

- Programa P2: Crea 4 autors i 4 llibres amb diferents quantitats d'autoria per a cada llibre.
- Programa P3:
 - Mostra els llibres amb els seus autors, cosa fàcil doncs Llibre incorpora la llista d'autors.
 - Mostra els autors amb els seus llibres, cosa que necessita una consulta doncs Autor no accedeix als llibres.

I per fer aquesta consulta JPQL, observar el JOIN que enllaça un FitxaLlibre amb els seus Autor.

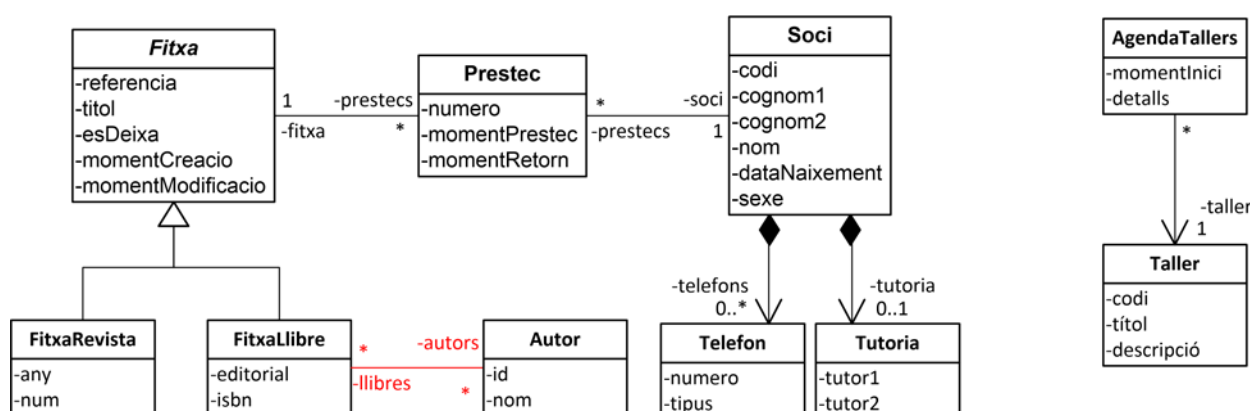
Com es comenta en el codi, aquesta consulta seria més fàcil i eficient si la parametritzem.
- Programes P4 i P5:
 - Mateix programa P3 però amb la consulta JPQL parametritzada.
 - P4: Passem per paràmetre l'identificador d'autor.
 - P5: Passem per paràmetre l'autor sencer => Funciona per què Autor té correctament definit equals.

Atenció! Els diferents tipus de CascadeType existents, aplicats en una relació ManyToMany, afecten a la classe apuntada. És a dir, si dins FitxaLlibre, el camp autors el marquem amb CascadeType.REMOVE, provocaria que l'eliminació d'un objecte FitxaLlibre provocaria l'eliminació dels seus autors (no confondre amb l'eliminació de les files dins la taula pont AUTORIA)

Continuació pràctica – Relacions ManyToMany bidireccional (Autoria entre Llibre i Autor)

17/02/25

Considerar la versió 6 del disseny UML:



que incorpora (en vermell) la relació bidireccional entre FitxaLlibre i Autor (que ja teníem però unidireccional)

Passar d'una ManyToMany **unidireccional** a una ManyToMany **bidireccional** és molt-molt-molt senzill, doncs simplement cal:



- Incorporar la col·lecció a la classe que no feia referència a l'altra classe (en el nostre cas `Autor`) i els mètodes `add`, `remove` i `iterator` que correspongui, **mantenint la sincronització entre les classes**.
- Marcar-la amb la marca `ManyToMany` amb `mappedBy` (tota la definició de la taula ja és a l'altra classe).

Projectes:

- 250217_5_BibliotecaV6
- 250217_6_BibliotecaV6A
- 250217_7_BibliotecaV6A_Hibernate
- 250217_8_BibliotecaV6X_Hibernate

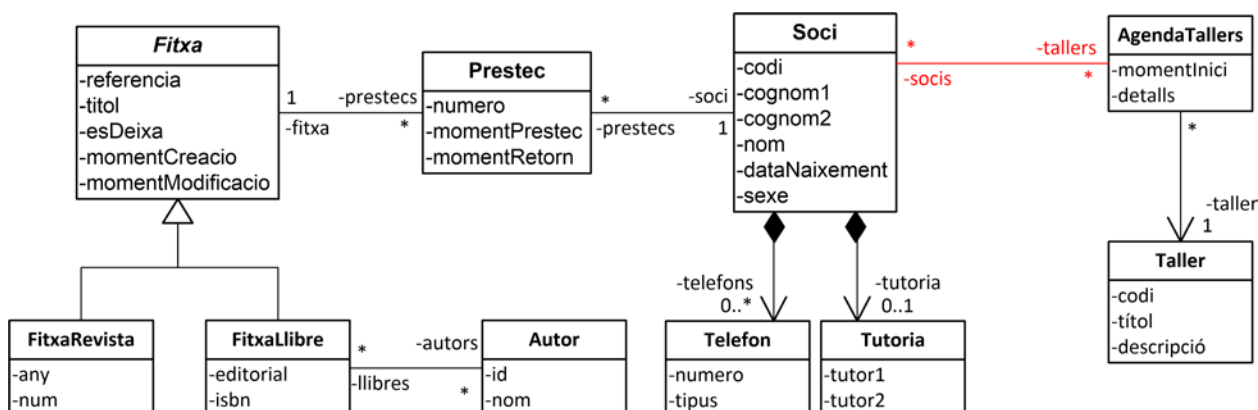
Respecte els projectes:

- Observar que s'ha completat la classe `Autor` afegint la llista de llibres i els mètodes `addLlibre`, `removeLlibre` i `iteLlibre`.
- Observar l'anotació `ManyToMany` a `Autor` en projecte `BibliotecaV6A` i la corresponent marca `many-to-many` a `Autor` en `autor.xml` de projecte `BibliotecaV6X_Hibernate`.
- Els mètodes `addLlibre` i `removeLlibre` de la classe `Autor` estan sincronitzats amb els mètodes `addAutor` i `removeAutor` de `FitxaLlibre`.
- Programa P2: Crea 4 autors i 4 llibres amb diferents quantitats d'autoria per a cada llibre.
- Programa P3:
 - Mostra els llibres amb els seus autors, cosa fàcil doncs `Llibre` incorpora la llista d'autors.
 - Mostra els autors amb els seus llibres, cosa fàcil doncs `Autor` incorpora la llista dels llibres.

Exercici per dimecres, 19 de febrer - Continuació pràctica – Pràctica de relacions `ManyToMany`

19/02/25

Considerar la versió 7 del disseny UML:



que incorpora (en vermell) la relació entre `Soci` i `AgendaTallers` per gestionar els tallers concrets que fa cada soci i quin són els socis que fan cada taller. Evidentment és una relació `ManyToMany`.

Evolucioneu el projecte `Biblioteca` (en versió Anotació i en versió marcatge XML) implementant aquesta relació, de manera que la taula a la BD que enregistri aquesta informació segueixi el disseny relacional:

```

INSCRIPCIO (#soci, #taller, #moment_inici)
on {soci} REFERENCIA SOCI
i {taller, moment_inici} REFERENCIA AGENDA_TALLERS
  
```

ATENCIÓ! El codi de soci es automàtic (gestionat per JPA). Això és un problema en el mètode `addSoci`, doncs podria ser que s'afegís un soci que encara no tingués codi (no marcat com a persistent) i la comprovació de l'existència del codi dins la llista s'efectua per codi. Per això cal comprovar, en el mètode `addSoci`, que el soci ja tingui codi per poder-lo afegir a la llista (és a dir, que ja sigui persistent).

Projectes solució:

- 250219_1_BibliotecaV7
- 250219_2_BibliotecaV7A
- 250219_3_BibliotecaV7A_Hibernate
- 250219_4_BibliotecaV7X_Hibernate

Respecte els projectes:

- Observar que s'ha completat la classe `AgendaTallers` afegint la llista de socis i els mètodes `addSoci`, `removeSoci` i `iteSocis`.
- Observar l'anotació `ManyToMany` a `AgendaTallers` en projecte `BibliotecaV7A` i la corresponent marca `many-to-many` a `AgendaTallers` en `taller.xml` de projecte `BibliotecaV7X_Hibernate`.
- Observar que s'ha completat la classe `Soci` afegint la llista de tallers (objectes `AgendaTallers`) i els mètodes `addTaller`, `removeTaller` i `iteTallers`.
- Observar l'anotació `ManyToMany` a `Soci` en projecte `BibliotecaV7A` i la corresponent marca `many-to-many` a `Soci` en `soci.xml` de projecte `BibliotecaV7X_Hibernate`.
- Els mètodes `addSoci` i `removeSoci` de la classe `AgendaTallers` estan sincronitzats amb els mètodes `addTaller` i `removeTaller` de `Soci`.
- Programa P2: Crea 4 socis i 1 taller amb 4 realitzacions i diverses inscripcions dels socis.
- Programa P3:
 - Mostra les realitzacions del taller amb les inscripcions de socis.
 - Mostra els socis amb les seves inscripcions

Més mètodes per gestionar objectes: detach, merge, refresh i remove

19/02/25

- a) `em.contains(obj)`
Per esbrinar si l'`EntityManager` està controlant l'objecte `obj`.
- b) `em.detach(obj)`
Allibera un objecte de l'`EntityManager`, de manera que aquest ja no el controla.
- c) `obj2 = em.merge(obj1)`
Si l'`EntityManager` està controlant l'objecte `obj1`:
 ⇒ No té efecte. Retorna el mateix objecte
 Si l'`EntityManager` no controla l'objecte `obj1` però en controla un altre amb mateix ID que `obj1`:
 ⇒ Retorna la referència a l'objecte ja controlat però modificat amb els continguts de `obj1`.
 Si l'`EntityManager` no controla l'objecte `obj1` i tampoc controla cap objecte amb mateix ID que `obj1`:
 ⇒ Retorna una referència a un nou objecte persistent, calcat a `obj1`, però NO és `obj1`.
- d) `em.refresh(obj)`
Refresca un objecte gestionat per l'`EntityManager` amb les dades existents a la BD (per si algun altre procés hagués efectuat canvis a l'objecte a la BD)
- e) `em.remove(obj)`
Marca un objecte persistent per a ser eliminat de la BD. Continua existent en memòria, però no controlat per EM. El canvi passarà a la BD en fer un `flush()` o `commit()`. En cas de `flush()` s'eliminarà de la BD si s'arriba a efectuar `commit()`.
- Projectes exemple: `250219_5_BibliotecaV7A_Hibernate_ProvesPersistenciaObjectes`
- Programa P02 amb proves de `merge`
 - Programa P03 amb proves de `refresh` i `remove`

Esdeveniments (triggers) en JPA

19/02/25

[JPA permet programar esdeveniments](#) sobre una classe, per indicar accions a executar abans o després d'altres accions. Disposem de: /

`@PostLoad / post-load`
`@PrePersist / pre-persist`
`@PostPersist / post-persist`
`@PreUpdate / pre-update`
`@PostUpdate / post-update`
`@PreRemove / pre-remove`
`@PostRemove / post-remove`

En [aquest enllaç](#) es veuen les 2 opcions.

- Cas en que tenim accés al codi.

La classe `Employee` incorpora dos mètodes (anomenats `prePersist` i `preUpdate`, però que podrien tenir qualsevol nom), precedits de l'anotació corresponent (que és el que provoca la seva execució).

Si es té accés al codi per incorporar els mètodes però estem usant marcatge XML, dins el marcatge de la classe inclourem els elements `<pre-... >` i `<post-... >` que corresponguin en ubicació segons `orm_3_1.xsd`.

- Cas en que no tenim accés al codi.

Dissenyem una nova classe (`EmployeeEventListener` a l'exemple) que incorpora els mètodes.

Cal indicar a la classe on cal invocar els mètodes (`Employee` a l'exemple), quina és la classe que conté els mètodes:

- En el cas d'anotacions, via `@EntityListeners` abans de la definició de la classe.
- En el cas de marcatge XML, via element `<entity-listeners>` ubicat segons `orm_3_1.xsd`.

Consultes en JPQL: consultes JPA i consultes natives SQL

19/02/25

Info JPQL: https://en.wikibooks.org/wiki/Java_Persistence/JPQL

Funcions usables en JPQL: https://en.wikibooks.org/wiki/Java_Persistence/JPQL#Functions

JPQL permet 4 tipus de consultes:

- Normals, sense paràmetres
- Amb paràmetres
- Amb nom (sense o amb paràmetres)
- Natives SQL

Projectes exemple:

- 250219_6_BibliotecaV7
- 250219_7_BibliotecaV7A
- 250219_8_BibliotecaV7A_Hibernate
- 250219_9_BibliotecaV7X_Hibernate

Respecte els projectes:

- Programa prova P03: Inclou exemples de consultes JPQL normals no parametritzades. [Vídeo](#)
- Programa prova P04: Inclou exemples de consultes JPQL parametritzades. [Vídeo](#)
- Programa prova P05: Inclou exemples de consultes JPQL amb nom: [Vídeo](#)
S'acostumen a incloure a la classe principal sobre la que operen, però poden estar en qualsevol classe. Observeu dues consultes amb nom a la classe `Soci`, amb la sintaxi quan s'incorporen via anotacions (abans de la classe) o via XML (després de l'element `<table>` i abans de l'element `<attributes>`)
- Programa prova P06: Inclou exemples de consultes natives SQL: [Vídeo](#)

Atenció: **EclipseLink és més restrictiu / Hibernate és més permissiu** en la sintaxi de les consultes. Exemples:

- Hibernate permet fer una consulta com `select count(*) from Soci s;`
EclipseLink davant la consulta anterior llença excepció similar a:

```
[13, 13] The left expression is missing from the arithmetic expression.
[14, 14] The right expression is missing from the arithmetic expression.
```

 Senzillament no li agrada l'asterisc. Cal escriure: `select count(s) from Soci s;`
- Hibernate permet en alguns llocs no posar àlies: `select s from Soci where sexe = 'F';`
EclipseLink obliga a usar sempre l'àlies: `select s from Soci s where s.sexe = 'F';`

Instruccions NO SELECT en JPQL: instruccions JPA (update/delete) i instruccions natives SQL

20/02/25

De manera similar a XQJ (M06-UF3) i JDBC (M03-UF6), JPQL distingeix entre:

- Instruccions JPQL per QL (com les `SELECT` de SQL)
- Instruccions JPQL per DML (no QL) i DDL-DCL (com les `UPDATE`, `DELETE`, `INSERT`, `CREATE`, `ALTER` de SQL)

Recordem (M06-UF3) que en XQJ, el programador controla com actuar segons el tipus d'instrucció:

- Si és QL, s'obté un `XQResultSequence` amb la informació, en executar:
 - `xqe.executeQuery(consulta)` on `xqe` es un `XQExpression` en consultes no parametritzades
 - `xqpe.executeQuery()` on `xqpe` és un `XQPreparedExpression` que conté la consulta parametritzada.
- Si no és QL, (DML o no DML), intenta fer el què es demana, sense donar resultat, en executar:
 - `xqe.executeCommand(instrucció)` on `xqe` es un `XQExpression` en instrucció no parametritzada
 - No existeix `executeCommand` per instruccions parametritzades.Recordem que en llenguatge XQUF, les instruccions DML són considerades consultes i s'executen amb `executeQuery` (SGBD-XML BaseX i Oracle)

Recordem (M03-UF6) que en JDBC, el programador controla com actuar segons el tipus d'instrucció:

- Si és QL, s'obté un `ResultSet` amb la informació, en executar:
 - `st.executeQuery(consulta)` on `st` es un `Statement` en consultes no parametritzades
 - `ps.executeQuery()` on `ps` és un `PreparedStatement` que conté la consulta parametritzada.
- Si no és QL, s'obté el nombre de files gestionades (DML) o zero (no DML), en executar:
 - `st.executeUpdate(instrucció)` on `st` es un `Statement` en consultes no parametritzades
 - `ps.executeUpdate()` on `ps` és un `PreparedStatement` que conté la instrucció parametritzada.

Vegem-ho en JPQL:

Anteriorment hem vist que per executar una `SELECT`, es procedeix a crear una `Query q` amb diferents possibilitats:

- `q=em.createQuery(consulta)` per una consulta JPA (amb o sense paràmetres)
- `q=em.createNamedQuery(nomConsulta)` per una consulta JPA amb nom (amb o sense paràmetres)
- `q=em.createNativeQuery(consulta)` per una consulta SQL tradicional (amb o sense paràmetres).

En qualsevol cas, la consulta s'executa amb els mètodes `q.getResultList` o `q.getSingleResult`.

Per executar instruccions `NO SELECT`, es crearà igualment una `Query q` amb les mateixes 3 possibilitats anteriors, però la instrucció, enlloc de ser una `SELECT`, serà una instrucció:

- `DELETE/UPDATE` per eliminar/modificar instàncies d'una classe, en cas de `Query` o `NamedQuery` gestionades per JPA.
- Qualsevol instrucció SQL `NO SELECT` en cas de `NativeQuery` gestionada directament pel SGBD via JDBC

I la diferència està en com executar-la. Enlloc d'usar els mètodes `q.getResultList` o `q.getSingleResult`, usarem `q.executeUpdate()` que retorna:

- Si és una `Query` o `NamedQuery` de JPA, el nombre d'instàncies eliminades/modificades.
- Si és una instrucció SQL via `NativeQuery`, el nombre de files afectades o zero si és instrucció DDL-DCL (com en JDBC)

En qualsevol cas (`Query`, `NamedQuery` o `NativeQuery`), cal tenir una transacció oberta i cal efectuar `commit` per validar els canvis o `rollback` per desfer-los.

Recordeu que una instrucció SQL-DDL provoca un `commit` automàtic a la BD i inici de nova transacció. Per tant, si s'executa una SQL-DDL via `NativeQuery`, s'haurà produït un `commit` automàtic... Per cert... millor no executar instruccions SQL-DDL des dels programes...

ATENCIÓ: L'`EntityManager` NO s'assabenta dels canvis en les instàncies que està gestionant davant instruccions JPQL `DELETE/UPDATE` (i per suposat via instruccions SQL natives)

- Si es sospita que algun objecte s'ha modificat a la BD (via JPQL/consulta nativa), caldrà executar un `refresh`
- Si es sospita que algun objecte s'ha eliminat de la BD (via JPQL/consulta nativa), en cas d'executar un `refresh` es produirà excepció indicant que no es troba cap fila amb l'identificador de l'objecte i la solució passa per eliminar l'objecte de l'`EntityManager` via `detach`.

ATENCIÓ en executar instrucció `delete` amb condició de filtre sobre classe relacionada via `ManyToMany`. En una relació `ManyToMany` entre dues classes A i B, sempre que s'elimina un objecte d'A, s'eliminen automàticament les

relacions que pugui tenir amb objectes de B, encara que a nivell de BD la FK de la taula pont no tingui eliminació en cascada. JPA s'encarrega d'eliminar primer (1) les files de la taula pont i després (2) la fila corresponent a l'objecte A. Per tant, si pel filtre es necessita passar per la taula pont, funcionarà en el pas (1), però en aplicar el mateix filtre en el pas (2) ja no filtrarà, doncs ja no existeixen les files de la taula pont. **Compte, compte, compte!!!**

Projecte amb exemples:

- 250220_1_BibliotecaV7A_Hibernate_ProvesInstruccionsNoSelect

En aquest projecte, els programes P03-P04-P05-P06 criden abans P01-P02 per què tots ells necessiten tenir la BD amb les dades que emplena P02 i així s'evita que l'alumne hagi d'executar P01-P02 manualment.

- Programa P01: Crea taules
- Programa P02: Insereix uns quants tallers amb agències, socis, inscripcions, fitxes i préstecs.
- Programa P03: JPQL DELETE – Primer executa P01-P02 per assegurar dades originals
 - Elimina socis amb cognom>'P' i un d'ells té inscripcions, que les elimina prèviament.
 - Intentem eliminar un taller que té agendes, i el programa PETA. L'error que reporta Hibernate és d'Oracle: **s'ha violat la restricció d'integritat (M06UF2.AGENDA_TALLER_FK_TALLER) - s'ha trobat un registre fill**
 - Intentem eliminar un soci que té préstecs, i el programa PETA. L'error que reporta Hibernate és d'Oracle: **s'ha violat la restricció d'integritat (M06UF2.PRESTEC_FK_SOCIO) - s'ha trobat un registre fill**

Resum:

- En intentar eliminar un Soci que té agendes, JPA executa delete previ de les agendes i telèfons.
- En intentar eliminar un Soci que té préstecs, JPA NO executa delete previ dels préstecs i PETA per FK.
- En intentar eliminar un Taller amb agendes, JPA NO executa delete previ d'agendes i PETA per FK.

Per què “sembla” que en unes ocasions funciona i en altres no ho fa?

En primer lloc, ha de quedar clar que:

CascadeType.REMOVE no afecta a sentències delete de JPQL. Només afecta al mètode remove.

Explicació:

- La relació entre Soci i AgendaTallers, que genera la taula INSCRIPCIO, és una relació ManyToMany entre dues entitats. NO hi ha una entitat Inscripcio.

En una relació ManyToMany entre dues classes A i B, sempre que s'elimina un objecte d'A, s'eliminen automàticament les relacions que pugui tenir amb objectes de B, encara que a nivell de BD la FK de la taula pont no tingui eliminació en cascada. JPA s'encarrega d'eliminar primer les files de la taula pont i després la fila corresponent a l'objecte A. En Odoó és igual!

Fixem-nos que elimina les relacions que puguin existir amb objectes de l'altra classe B, però **NO elimina** els objectes de B!!!

Si la relació entre les classes A i B hagués necessitat d'una classe associativa AB, llavors estaríem parlant de tres Entity, amb:

- Relacions ManyToOne de AB cap a A i de AB cap a B
- Relacions OneToMany de A cap a AB i de B cap a AB

En aquest cas no hi hauria eliminació automàtica dels objectes relacionats. Caldria explicitar eliminació prèvia o que existís `on delete cascade` en la definició de la FK.

- La relació de Soci cap a Prestec és una OneToMany i en aquest cas, l'eliminació de soci via delete de JPQL NO provoca l'eliminació dels seus préstecs. Caldria explicitar eliminació prèvia o que existís `on delete cascade` en la definició de la FK.
- De Taller cap a AgendaTallers NO hi ha relació (és de AgendaTallers cap a Taller) i per tant, és impossible que l'eliminació d'un taller provoqui l'eliminació de les seves agendes. Caldria explicitar eliminació prèvia o que existís `on delete cascade` en la definició de la FK.
- Programa P04: JPQL DELETE – Primer executa P01-P02 per assegurar dades originals

Reedició del programa P03, sent el programador qui:

- Abans d'eliminar taller amb agendes, es preocupa d'eliminar les agendes:

```
delete from AgendaTallers at where at.taller.codi='LI01'
```

```
delete from Taller t where t.codi='LI01'
```
- Abans d'eliminar soci amb préstecs, es preocupa d'eliminar els préstecs.

```
delete from Prestec p
```

```
where p in (select xxx from Soci s join s.prestecs xxx
```

```
where s.cognom1='Mendez')
```

Atenció! Per accedir a una col·lecció en una SELECT, cal fer un JOIN com es mostra en instrucció prèvia.

```
delete from Soci s where s.cognom1='Mendez'
```

- Programa P05: Instruccions DML natives – Primer executa P01-P02 per assegurar dades originals originals. Mireu els comentaris que conté.
- Programa P06: Comprovació que l'EntityManager no s'assabenta de les modificacions/eliminacions dels objectes que està gestionant. Primer executa P01-P02 per assegurar dades originals

El programa mostra missatges explicatius del què està succeint en cada moment.

S'utilitza `refresh` per refrescar canvis a la BD i usariem `detach` per desvincular objectes de l'EM.

Tipus d'accès de JPA a les dades: FIELD – PROPERTY

24/02/25

JPA pot gestionar el traspàs d'informació entre la BD i l'aplicació de dues maneres:

- Accés directe (camp de l'objecte ↔ camp de la taula): `AccessType.FIELD`
 És el mètode que hem utilitzat en tota la UF, fins aquesta darrera sessió, en la que veurem que en ocasions pot ser necessari usar l'altre tipus d'accés. És similar al tipus d'accés que varem veure en JAXB (M06-UF1).
- Accés executant mètodes getter-setter: `AccessType.PROPERTY`
 - Mètode `getter` s'utilitza en el traspàs de camp de l'objecte → camp de taula
 - Mètode `setter` s'utilitza en el traspàs de camp de la taula → camp de la classe

Una classe pot incloure la marca `@Access/access` a nivell de classe, fet que afecta a totes les dades membre de la classe. Però si per alguna dada es pot canviar, es pot especificar en ella l'accés que interressi.

JPA no defineix un accés per defecte i pot dependre del proveïdor JPA.

- En anotacions: És usual que, si la classe no incorpora la marca `@Access`, s'utilitzi `FIELD` o `PROPERTY` en funció de la ubicació de la marca `@Id` (en la definició del camp o dels mètode `getter`) i llavors tota la resta de marques han de seguir la mateixa ubicació, excepte aquelles per les que s'especifiqui un `AccessType` diferent.

Però per evitar problemes, hagués estat millor marcar totes les classes amb `@Access (AccessType.FIELD)`

- En XML: Alguns proveïdors JPA permeten indicar un accés per defecte per a totes les classes, el qual es pot sobreesciure amb l'atribut `access` a nivell de cada classe.

En el nostre exemple, en tots els XML hem incorporat a cada `<entity class=... access="FIELD"...>`

Davant la inseguretat de l'accés per defecte segons el proveïdor JPA, **és altament recomanable utilitzar la marca `@Access/access` per a cada classe.**

Per últim, en casos en que per un camp s'assigni accés específic diferenciat de l'accés a nivell de classe, pot ser necessari que la dada/mètode que JPA hauria d'emprar segons l'accés a nivell de classe hagi de ser marcat amb `@Transient` per evitar possible duplictat de traspàs.

Exemple de necessitat de treballar amb tipus PROPERTY per algun camp:

Suposem que a la classe `Soci` li volem incorporar un camp `numTallers` que contingui el recompte d'inscripcions a tallers i que aquest camp NO sigui persistent a la BD. És un exemple no massa real... però imagineu una classe `C` on interressi tenir un camp `x` que es calcula a partir d'altres camps, és a dir, un camp calculat, i no es vol tenir persistent a la corresponent taula a la BD.

Tornem al nostre exemple no massa real. Afegim numTallers a Soci i no el volem a la BD però ha d'estar actualitzat en tot moment. Solució?

Projectes solució:

- 250224_1_BibliotecaV8
- 250224_2_BibliotecaV8A
- 250224_3_BibliotecaV8A_Hibernate
- 250224_4_BibliotecaV8X_Hibernate

Comentaris:

- La versió amb anotacions incorpora ja `@Access(AccessType.FIELD)` a totes les classes.
- La classe Soci incorpora el camp `int numTallers`, marcat com a `@Transient` (via anotacions) o amb etiqueta `<transient>` en fitxer `soci.xml`.
- Ens interessa incorporar mètode `getNumTallers` per poder-lo consultar, però `setNumTallers` no té cap sentit i no l'afegim.
- Com es gestiona `numTallers`? En el moment que s'afegeix un taller a `Soci.tallers`, `numTallers++` i quan s'elimina un taller de `Soci.tallers`, `numTallers--`.
- I quan es recupera un soci de la BD? En aquest cas, quan JPA s'encarrega de recuperar el contingut del camp `tallers` seria el moment d'aprofitar i ja que tenim la llista de tallers, emplenar `numTallers` amb la grandària de la llista `tallers`. Per tant, ens interessa dir a JPA que la càrrega del camp `tallers` no s'efectuï directament via `FIELD`, sinó via `PROPERTY` i hauré d'incorporar els camps `setTallers` i `getTallers`, que no els teníem, però que JPA utilitzarà... Els farem `private` si no interessa que un programador els pugui usar. Fixeu-vos com queda la classe `Soci` via anotacions:

```
@ManyToMany(mappedBy = "socis", cascade = CascadeType.PERSIST)
@Access(AccessType.PROPERTY)
private List<AgendaTallers> tallers = new ArrayList();

@Transient
private int numTallers;

public int getNumTallers() {
    return numTallers;
}

private List<AgendaTallers> getTallers() {
    return tallers;
}

private void setTallers(List<AgendaTallers> tallers) {
    this.tallers = tallers;
    this.numTallers = tallers.size();
}
```

En la versió amb marques, el codi Java és igual i cal retocar `soci.xml` amb:

```
<many-to-many name="tallers" mapped-by="socis" access="PROPERTY">
...
</many-to-many>
...
<transient name="numTallers"/>
```

- El programa P03 mostra com en recuperar els socis, el camp `numTallers` està correctament emplenat.