Informàtica

ICB0 Desenvolupament d'aplicacions multiplataforma

M06 Accés a dades

UF2 Persistència en BDR-BDOR-BDOO

Diari d'activitats

Isidre Guixà

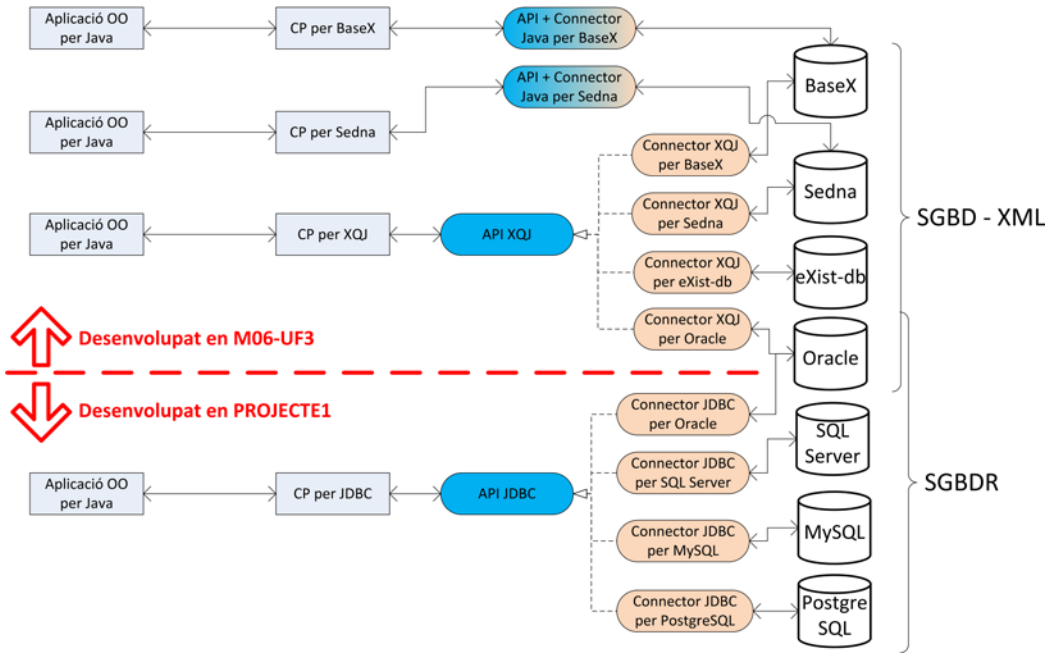
Curs 2022/23

Què sabem fer?	**02-12-22**

En UFs precedents hem practicat **el desenvolupament d'aplicacions OO que gestionen dades enregistrades en un SGBD amb l'objectiu final que l'aplicació OO sigui independent del SGBD** a emprar.

S'ha practicat usant el llenguatge Java, però també es pot efectuar en altres llenguatges pels que existeixi connectors adequats per a cada SGBD, ja sigui facilitats pel propi fabricant del SGBD o per tercers.

Per aconseguir que l'aplicació sigui independent del SGBD, cal desenvolupar una capa de persistència (CP) que contingui un conjunt complet de mètodes que s'encarreguin del diàleg amb el SGBD (consultes i actualitzacions) de manera que l'aplicació pugui ser única i invoqui, quan hagi de dialogar amb el SGBD, els mètodes de la CP que són els encarregats del diàleg. La imatge mostra els tipus de desenvolupaments practicats:



- En M06-UF3 hem desenvolupat:

- CP per a BaseX, utilitzant l'API i connector propietaris de BaseX. Aquesta capa només serveix per a connectar amb SGBD-XML BaseX.
- CP per a Sedna, utilitzant l'API i connector propietaris de Sedna. Aquesta capa només serveix per a connectar amb SGBD-XML Sedna.
- CP per a XQJ, que és una API (definició de funcionalitats) ideada per establir un protocol de connexió amb SGBD-XML i que permeti connectar amb qualsevol SGBD-XML que faciliti connector XQJ.

  Aquesta CP, en la fase d'implementació (escriptura de codi i compilació) només necessita de la llibreria on hi ha la definició de l'API (`xqjapi.jar`). En canvi, en temps d'execució, per connectar amb un SGBD-XML concret, precisa del connector que implementi l'API XQJ específic pel SGBD.

  A diferència de les CP anteriors que només servien per 1 SGBD, aquesta CP serveix per a qualsevol SGBD-XML que faciliti connector XQJ.

- En PROJECTE1 hem desenvolupat:

- CP per a JDBC, que és una API (definició de funcionalitats) ideada per establir un protocol de connexió amb SGBDR i que permeti connectar amb qualsevol SGBDR que faciliti connector JDBC.

  Aquesta CP, en la fase d'implementació (escriptura de codi i compilació) no necessita cap llibreria especial per què la definició de l'API està incorporada en JDK. En canvi, en temps d'execució, per connectar amb un SGBDR concret, precisa del connector que implementi l'API JDBC específic pel SGBD.

  De forma similar a la CP per a XQJ, aquesta CP serveix per a qualsevol SGBDR que faciliti connector JDBC.

Suposem que les diverses CP desenvolupades contenen el mateix conjunt de mètodes. Llavors, la problemàtica inicial a resoldre (aplicació OO independent de qualsevol SGBD) ja està assolida?

NO!!! A la imatge veiem que no hi ha 1 única aplicació, sinó 1 aplicació per a cada CP. La resolució completa l'assolirem a la UF4.

- En M06-UF3, el programa que utilitza la capa de persistència és idèntic en els 3 casos excepte en 3 punts:
  (1) Incorporació de la capa a la zona `import`:
  `import nomCapaPersistencia;`
  (2) Creació de l'objecte de la CP per invocar els mètodes de la capa:
  `nomCapaPersistencia cp = new nomCapaPersistencia(...);`
  (3) Utilització de la classe `Exception` que correspongui segons capa de persistència usada.

  En conseqüència l'aplicació està lligada a la CP; no està lligada al SGBD però de moment continua lligada a la CP. La independència total, també de la CP, l'assolirem a la UF4.

  De la mateixa manera que les CP desenvolupades en M06-UF3 gestionaven dades d'una empresa (classes `Empresa`- `Departament`-`Empleat`) emmagatzemades en SGBD-XML en format XML seguint específics DTD, podríem desenvolupar una CP amb idèntics mètodes gestionant dades emmagatzemades en SGBDR via JDBC. El programa seria el mateix; només caldria efectuar els canvis dels 3 punts anteriors.

- En PROJECTE1, passa el mateix via JDBC. L'aplicació desenvolupada incorpora els 3 punts anteriors.

  De la mateixa manera que s'ha desenvolupat la CP per JDBC, es podria definir un format XML per emmagatzemar les dades de les classe gestionades en PROJECTE1 i es podria desenvolupar:
  - Una CP amb idèntics mètodes gestionant dades emmagatzemades en SGBD-XML, usant l'API XQJ.
  - Una CP amb idèntics mètodes gestionant dades emmagatzemades en BaseX, usant l'API pròpia.
  - Una CP amb idèntics mètodes gestionant dades emmagatzemades en Sedna, usant l'API pròpia.

  L'aplicació seria la mateixa; només caldria efectuar els canvis dels 3 punts anteriors.

La imatge distingeix amb color blau les API (definició de funcionalitats) que ha d'invocar la CP i amb color taronja els connectors (classes concretes que implementen l'API). Els connectors es necessiten en temps d'execució, per establir la connexió amb el SGBD que pertoqui. Les API són necessàries en temps d'implementació.

En el cas de les APIs específiques/propietàries (per exemple BaseX i Sedna) és força comú que API i classes estiguin en una mateixa llibreria, però en les APIs estàndards (XQJ i JDBC) sempre estan separades, doncs per una banda existeix la definició de l'API, elaborada per l'organisme/grup d'experts que pertoqui i per altra banda existeix la implementació concreta de les classes, específica per a cada SGBD.

Què farem a la UF2?	**02/12/22**

El programa oficial de la UF2 incorpora:

- Implementació de capes de persistència per a BDR-BDOR-BDOO
- Eines de mapatge ORM

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:

- Amb el desenvolupament de capes de persistència efectuat a la UF3 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.

Per aquest motiu, en el M&F **la UF2 es centra en l'accés a dades via eines ORM**.

Eines de mapatge objecte-relacional (ORM) – Introducció - Dossier de l'IOC (només ORM)	**13-12-22**

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.

Actualitat de les versions de JDO – JPA (no instal·leu cap de les versions que indica el dossier)

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.

Versions estables que suporten JPA 2.2 existents en el moment de començar aquesta UF en el curs 22/23:
- Versió estable d'Hibernate: 5.6.14 (4/11/22) que compleix estàndard JPA 2.2 i **requereix JDK8/11/17/18**
- Versió estable d'EclipseLink: 2.7.11 (10/08/22) que compleix l'estàndard JPA 2.2 i **requereix JDK8**

L'octubre de 2020 s'ha publicat l'estàndard Jakarta Persistence 3.0 que forma part de Jakarta EE, que és l'evolució *open source* de Java EE, propietat d'Oracle.
- Les versions 3.x i 4.x d'EclipseLink donen suport a Jakarta Persistence 3.0. Cal **JDK11/17**
- Les versions 5.6.x d'Hibernate són compatible amb JPA 2.2 i Jakarta Java 3.0. Cal **JDK8/11/17/18**
- Les versions 6.x d'Hibernate són compatibles amb Jakarta Java 3.1 i 3.0. **Cal JDK11/17/18**

Si comparem la documentació de JPA 2..2 amb la documentació de Jakarta Persistence 3.0, entre altres canvis, observem diferència en els noms dels paquets:

JPA 2.2	Jakarta Persistence 3.0
`javax.persistence`	`jakarta.persistence`
`javax.persistence.criteria`	`jakarta.persistence.criteria`
`javax.persistence.metamodel`	`jakarta.persistence.metamodel`
`javax.persistence.spi`	`jakarta.persistence.spi`

DataNucleus (JPOX) és un producte OpenSource que compleix els 2 estàndards:
- JPA 2.2+ / Jakarta 3.0+
- JDO 3,2+

Tots aquests productes, a més d'ORM, faciliten solucions per altres tipologies de SGBD!!! Més informació a:

- JPA vs Hibernate
- Java Persistence API (JPA)

Productes JPA2.2 dels que utilitzarem en el curs 22/23 el connector JPA:

- Hibernate 5.6.14. S'aconsella crear a NetBeans la biblioteca amb nom
  `Hibernate 5.6.14`
  ja que els projectes solució que es facilitaran en aquest dossier incorporaran una biblioteca amb aquest nom.

  El dossier de l'IOC explica que les llibreries que cal carregar són les que hi ha dins la carpeta `jpa` i `required`. Això era així fins versions 5.1.x. A partir de 5.2, la carpeta `jpa` deixa d'existir.

  La nostra biblioteca ha de contenir les llibreries de la carpeta `lib/required` d'Hibernate 5.6.14.

  En 1/12/2022 l'enllaç anterior (definit a la web d'Hibernate i que porta a *SourceForge*) no funciona i deixa descarregar la versió 5.6.14. No ens podem descarregar tot l'Hibernate (no el necessitem) però com que només necessitem les llibreries requerides, tenim la sort de trobar-les aquí.

- EclipseLink 2.7.11. S'aconsella crear a NetBeans la biblioteca amb nom
  `EclipseLink 2.7.11`
  ja que els projectes solució que es facilitaran en aquest dossier incorporaran una biblioteca amb aquest nom.

  Aquesta biblioteca ha de contenir els jars següents d'EclipseLink 2.7.11:
  - `jlib/jpa/jakarta.persistence_2.2.3.jar`
  - `jlib/jpa/ org.eclipse.persistence.jpa.modelgen_2.7.11.v20220810-51b5b24bf1`
  - `jlib/jpa/ org.eclipse.persistence.jpars_2.7.11.v20220810-51b5b24bf1`
  - `jlib/eclipselink.jar`

  Fixem-nos que EclipseLink ja no proporciona `javax.persistence`... sinó `jakarta.persistence`..., cosa que succeeix des de que es va posar en marxa el projecte Jakarta. Però internament els noms dels paquets encara coincidien amb els de JPA2.2. Ha estat a partir de Jakarta 3 que s'ha canviat els noms dels paquets.

Compte! NetBeans incorpora algunes llibreries de JPA que no es corresponen amb les versions que utilitzarem en aquest curs. Per tant, cal descarregar els productes indicats i utilitzar els `jars` indicats.

En aquestes llibreries, introduir com a javadoc, el del projecte JPA 2.1/2.2 – JSR 338.

Peces bàsiques de JPA - Dossier de l'IOC (només ORM)	**15/12/22**
2.4.1. Entitats **2.4.2. El gestor d'entitats** **2.4.3. Unitats de persistència**	

Definició d'Unitat de Persistencia – Creació d'`EntityManager` — 15/03/22

- 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 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 què 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.

Exemple en NetBeans curs 22/23:

- Crear projecte Java nou i seleccionar el seu nom.
- `File | New File | Persistence | PersistenceUnit`

Observem que la versió que tenim de NetBeans en permet seleccionar EclipseLink (JPA 2.1) que no és el què ens interessa. Possiblement es podria configurar NetBeans per a JPA2.2...

Creació manual (nostre cas):

- Crear carpeta `META-INF` a l'arrel del projecte
- 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 JPA 2.2: **Capítol 8 de doc. oficial de JPA2.2 (8.2.1)**

```
<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">
```

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 és per EclipseLink (JPA 2.1) i no ens serveix. 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_2_2.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="javax.persistence.jdbc.url"
              value="URL jdbc per connectar amb SGBD"/>
    <property name="javax.persistence.jdbc.user" value="usuari"/>
    <property name="javax.persistence.jdbc.password" value="contrasenya"/>
    <property name="javax.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,...)

- En NetBeans curs 21-22, si `provider` és EclipseLink, en generar una `property` i escriure `name=`, mostra les propietats estàndards de JPA i les pròpies d'EclipseLink. En Hibernate, només JPA ☹!

- Dades per connectar amb Oracle (port habitual: 1521)
  Driver: `ojdbc8.jar` (ubicat a `pathOnEsOracle\dbhomeXE\jdbc\lib`)
  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 MySQL8 (aules Milà) (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 (servidor núvol) (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 MySQL`**, amb la llibreria adequada JDBC per connectar amb MySQL que correspongui

- Propietats específiques de cada proveïdor, com:
- `hibernate.dialect`, per indicar el llenguatge de comunicació 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 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 no és l'adequat, pot no generar adequadament les sentències.
  Per veure com emplenar aquesta variable, segons el SGBD a accedir, cal consultar la documentació (javadocs) incorporada en la versió d'Hibernate descarregada i cercar dins el paquet `org.hibernate.dialect` quina classe cal indicar com a valor de la propietat. Exemple:

  - Oracle21c: `org.hibernate.dialect.Oracle12cDialect`
  - PostgreSQL9.6 (servidor núvol): `org.hibernate.dialect.PostgreSQL96Dialect`
  - PostgreSQL10 (aules Milà): `org.hibernate.dialect.PostgreSQL10Dialect`
  - MySQL5 (servidor núvol): `org.hibernate.dialect.MySQL57Dialect`
  - MySQL8 (aules Milà): `org.hibernate.dialect.MySQL8Dialect`

- Altres que anirem incorporant a mida que es necessitin

Informació referent a les Unitats de Persistència dels projectes adjunts:

El fitxer `persistence.xml` incorpora tres UP:

- `UP-Oracle`, per connectar amb un Oracle de la VM
- `UP-PostgreSQL`, per connectar amb un PostgreSQL 10 (aules del Milà)
- `UP-MySQL`, per connectar amb un MySQL8 (aules del Milà)

Si voleu connectar amb un PostgreSQL i/o MySQL diferents, tingueu en compte dades a canviar.

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

Exemples de projectes amb diverses configuracions:

A continuació es presenten 6 projectes per fer diverses comprovacions. Tots ells contenen 3 configuracions d'execució, que invoquen el mateix programa passant per paràmetre el nom d'una unitat de persistència concreta (UP-Oracle, UP-MySQL, UP-PostgreSQL).

Proveu-los, com a mínim, en Oracle i en un PostgreSQL (el del núvol o un que hagueu instal·lat en M10). Penseu en adequar les dades de connexió de les unitats de persistència.

Comprovació 1: Projectes
- `221216_1_JPA_HibernateNomesLibJPA`
- `221216_2_JPA_EclipseLinkNomesLibJPA`

El programa només intenta crear l'`EntityManagerFactory`.

Els dos projectes només incorporen la llibreria JPA 2.2, és a dir, NO la implementació d'Hibernate o d'EclipseLink.

- En el cas d'Hibernate, la llibreria **`Hibernate 5.6.14 (Only JPA2.2)`** constituïda pel `javax.persistence-api-2.2.jar` de la carpeta `lib/required` d'Hibernate 5.6.14.
- En el cas d'EclipseLink, la llibreria **`EclipseLink 2.7.11 (Only JPA2.2)`** constituïda pel `jakarta.persistence_2.2.3.jar` de la carpeta `jlib/jpa` d'EclipseLink 2.7.11.

En ambdós casos, els programes es compilen, però a l'hora d'executar-los, en qualsevol SGBD, peten per què no tenen accés a la llibreria del corresponent proveïdor ORM (Hibernate / EclipseLink).

Comprovació 2: Projectes
- `221216_3_JPA_HibernateJPAComplet` amb llibreria **`Hibernate 5.6.14`**.
- `221216_4_JPA_EclipseLinkJPAComplet` amb llibreria **`EclipseLink 2.7.11`**

Mateix programa que els projectes anteriors, que només intenta crear `EntityManagerFactory`.

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 [oracle.jdbc.driver.OracleDriver]

  L'error diu que no troba la classe JDBC per connectar. Però si li afegim la llibreria, continuarà petant fins que no aconsegueix la connexió. És a dir, Hibernate estableix la connexió en el moment de crear la factoria.
- En EclipseLink sí té lloc la situació teòrica i la factoria es crea, sense intentar establir connexió.

Comprovació 3: Projectes
- `221216_5_JPA_HibernateJPA+JDBC`
- `221216_6_JPA_EclipseLinkJPA+JDBC`

El programa, a més de crear la factoria, també crea el gestor d'entitats. En aquest moment EclipseLink és quan estableix la connexió.

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 la informació subministrada 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 que donen aquestes eines acostuma a ser configurable via propietats específiques de les eines.

Com funciona JPA?	**20//12/22**

A l'apartat 2.2.1 del dossier 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 interessi 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 uniformitat.

Per tant, en tots els projectes que desenvolupem al llarg de la UF, haurem de tenir en compte com ens 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 JPA2.2	**20/12/22**

- Obligatorietat de constructor sense paràmetres, públic o protegit
- `Serializable`
- No pot ser 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	**20/12/22**

- Primeres anotacions obligatòries
  - Nivell de classe: `@Entity,` per indicar que és una classe persistent
  - Nivell de dades: `@Id,` 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.

  Projectes amb la classe `ProvaTipusDades` amb configuració mínima:
  `221220_1_ProvesTraduccióTipusDades_Hibernate`
  `221220_2_ProvesTraduccióTipusDades_EclipseLink`
  La versió per EclipseLink necessita la `property` `eclipselink.canonicalmodel.subpackage` (situació estranya)

- La carpeta `META-INF` amb el fitxer `persistence.xml` 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 `<class>` (en ubicació segons indiqui `persistence_2_2.xsd`)

  És possible que algun proveïdor de persistència, sense tenir la informació de les classes marcades, incorpori directament aquelles que contenen alguna marca.

- Les UP subministrades contenen les propietats següents que cal que inicialment deixem comentades:
  De JPA: `javax.persistence.schema-generation.database.action`
  Pròpia d'Hibernate: `hibernate.hbm2ddl.auto`
  Pròpia d'EclipseLink: `eclipselink.ddl-generation`

- **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 `property`:
  `javax.persistence.schema-generation.database.action`
  que permet opcions: `none, create, drop-and-create, drop`
  (cercar aquesta propietat en PDF de documentació oficial JPA2.2)

  Executem el programa de prova comprovant què passa segons proveu amb `create`, `drop-and-create`,...
  Mirem la taula que es crea a la BD.

  Alerta!!! Perfecte l'opció `create` en la primera ocasió si volem que creï les taules i `drop-and-create` en següents, mentre s'afina la definició de les anotacions, però en explotació: `none`!!!

  La majoria d'ocasions, la creació de la BD no l'efectua l'aplicació sinó que s'efectua prèviament via guions.

  Alguns proveïdors de persistència faciliten propietat alternativa amb més possibilitats:

  - Hibernate: `hibernate.hbm2ddl.auto` amb els valors:
  ✓ `validate`: validate the schema, makes no changes to the database (en explotació)
  ✓ `update`: update the schema (en desenvolupament) **=> Per treballar a classe**
  ✓ `create`: creates the schema, destroying previous data.
  ✓ `create-drop`: drop the schema at the end of the session.

  - EclipseLink: `eclipselink.ddl-generation` amb els valors:
  ✓ `create-tables`
  ✓ `create-or-extend-tables`: Similar a `update` d'Hibernate **=> Per treballar a classe**
  ✓ `drop-and-create-tables`
  ✓ `none`

- **Comprovació 3**: Desactivar la `property`
  `javax.persistence.schema-generation.database.action`
  i activar la `property` `hibernate.hbm2ddl.auto` en Hibernate
  o la `property` `eclipselink.ddl-generation` en EclipseLink.

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

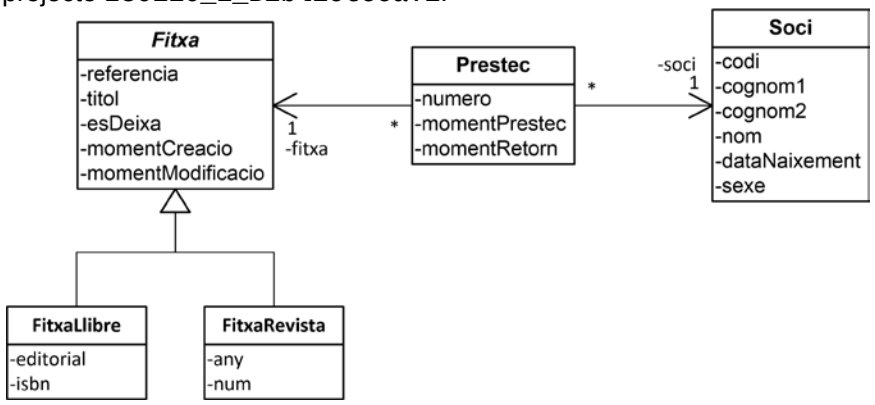
- En l'execució dels projectes anteriors, deixant que el proveïdor JPA creï la taula, observem:

Tipus de dada	Hibernate			EclipseLink		
	Oracle	PostgreSQL	MySQL	Oracle	PostgreSQL	MySQL
Integer	NUMBER(10)	Integer	int(11)	NUMBER(10)	Integer	int(11)
BigDecimal	NUMBER(19,2)	numeric(19,2)	decimal(19,2)	NUMBER(38)	numeric(38,0)	decimal(38,0)
BigInteger	NUMBER(19,2)	numeric(19,2)	decimal(19,2)	NUMBER(38)	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 CHAR)	character(1)	char(1)	CHAR(1)	character(1)	char(1)
Double	FLOAT(126)	double precision	double	NUMBER(19,4)	double precision	Double
Float	FLOAT(126)	Real	Float	NUMBER(19,4)	double precision	Float
Long	NUMBER(19)	Bigint	bigint(20)	NUMBER(19)	Bigint	bigint(20)
Short	NUMBER(5)	Smallint	smallint(6)	NUMBER(5)	Smallint	smallint(6)
String	VARCHAR2(255 CHAR)	character varying(255)	varchar(255)	VARCHAR2(255)	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.

Pràctica	**10/01/23**

Es vol assolir i gestionar la persistència de les classes del següent UML, de les que ja es facilita una implementació en el projecte `230110_1_BibliotecaV1`.



Pràctiques – Nomenclatura	**12/01/23**

Efectuarem l'aprenentatge a partir d'un projecte inicial `230110_1_BibliotecaV1`, que anirà evolucionant.

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`,

Si en algun moment es decideix fer algun canvi a les classes de `Biblioteca`, passarem a `BibliotecaV2` i així successivament.

Primeres anotacions – Primers mètodes per gestionar els objectes – Classe `Soci`	**12/01/23** **17/01/23**

- Anotacions a incorporar
  - Nivell de classe: `@Table`
  - Nivell de dades: `@Basic` | `@Column`
- **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 JPA2.2 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 5.6.14: Genera excepció en enregistrar els canvis a la BD (`flush` o `commit`).
    - ✓ EclipseLink 2.7.11: Genera excepció en enregistrar 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:**

  `230112_1_BibliotecaV1` (projecte "net" sense cap anotació i amb els requeriments a nivell de classe per a `Soci`: `Serializable`, constructor sense paràmetres, NO `final`, camps identificadors immutables)

`230112_2_BibliotecaV1A` (projecte anterior amb les anotacions)
Observar que necessita, per compilar, el jar de JPA 2.2 (indiferent d'Hibernate o EclipseLink)
`230112_3_BibliotecaV1A_Hibernate` i `230112_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)
`230117_1_BibliotecaV1X_Hibernate` i `230117_2_BibliotecaV1X_EclipseLink`, de proves amb marcatge XML.

Observar que la carpeta META-INF està SEMPRE 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_2_2.xsd` adjunt.

Capçalera del fitxer XML que conté el marcatge d'una 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 que conté el marcatge d'una 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 de mapatge per JPA 2.2: **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">
```

⇨ 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.

Vídeo del desenvolupament

ATENCIÓ!!! Actuació de JDBC en tancar connexió amb transacció oberta	**17/01/23**

- 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();
}
```

Exercici per dijous, 19 de gener - Continuació pràctica – Classe `Fitxa`	**19/01/23**

- 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:
  - Un índex per títol
  - Les fitxes s'identifiquen per la seva referència
  - 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 retoquem classe `Fitxa` amb els requeriments mínims per poder-la fer persistent
- `230119_1_BibliotecaV1`
- `230119_2_BibliotecaV1A`
- `230119_3_BibliotecaV1A_Hibernate`
- `230119_4_BibliotecaV1X_Hibernate`

Vídeo de la solució

Continuació pràctica – Relacions `ManyToOne` – Clàusules `fetch` i `cascade` - Classe `Prestec`	**19/01/23**

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
```

Més endavant treballarem la resta de relacions. En totes elles, 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 els 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` en els camps relacionals permet decidir l'actuació que es vol:
  `FetchType.LAZY` (actuació gandula-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 (cosa que no hem utilitzat en els exemples precedents) 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-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:

`CascadeType.PERSIST`	`CascadeType.REFRESH`	`CascadeType.DETACH`
`CascadeType.MERGE`	`CascadeType.REMOVE`	`CascadeType.ALL` (les engloba totes)

Projectes solució:

- En primer lloc retoquem classe `Prestec` amb els requeriments mínims per poder-la fer persistent
- `230119_5_BibliotecaV1`
- `230119_6_BibliotecaV1A`
- `230119_7_BibliotecaV1A_Hibernate`
- `230119_8_BibliotecaV1X_Hibernate`

Vídeo del desenvolupament

Atenció!!! Hibernate no actua correctament davant Fetch.LAZY?

El programa P2 dels projectes anteriors crea el soci 300, la referència R111111111 i el préstec 1000.
El programa P3 cerca el soci 300 i posteriorment el préstec 1000.

- Activeu depuració i atureu-vos en la cerca del soci 300.
- Troba el soci i el mostra per pantalla
- Posteriorment cerca el préstec 1000, que és del soci 300 i la fitxa R111111111
- Comprovem que efectua `SELECT from PRESTEC` i... `SELECT from FITXA...` quan aquesta darrera cerca no s'hauria d'efectuar per què el camp fitxa està marcat amb `Fetch.LAZY`. No executa `SELECT from SOCI...` doncs ja el te carregat en memòria.
- Unes instruccions més enllà s'executa `System.out.println(p)`, que invoca el mètode `toString` de `Prestec` i en aquest moment és quan necessita la fitxa i hauria de provocar la `SELECT` per carregar la fitxa en memòria.

Hibernate no actua correctament davant `Fetch.LAZY`?

L'especificació de JPA 2.2, a l'apartat 11.1.30, respecte l'anotació `ManyToOne` explicita:

> *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.*

És a dir, el proveïdor de la persistència (Hibernate en el nostre cas) no està obligat a seguir l'estratègia LAZY. No he trobat en la documentació d'Hibernate, si la segueix o no la segueix pels camps `ManyToOne`, però a títol informatiu, respecte els camps Basic, detalla:

> *fetch - FetchType (defaults to EAGER)*
>
> *Defines whether this attribute should be fetched eagerly or lazily. JPA says that EAGER is a requirement to the provider (Hibernate) that the value should be fetched when the owner is fetched, while LAZY is merely a hint that the value is fetched when the attribute is accessed. Hibernate ignores this setting for basic types unless you are using bytecode enhancement. See the Bytecode Enhancement for additional information on fetching and on bytecode enhancement.*

Sembla que pels camps `Basic` no te en compte l'estratègia `LAZY` a no ser que s'utilitzi la tècnica *Bytecode Enhancement* que ignorem.

Per tant... la causa del nostre problema radica en que Hibernate no suporti `LAZY` pels camps `ManyToOne`? La resposta és NO i la causa radica en què no hem fet cas dels ***warnings*** que Hibernate ens deixava anar cada vegada que creàvem l'`Entity Manager`, relatius a que no era bo tenir mètodes `setter/getter final`.

Eliminem tots els `final` dels mètodes `setter/getter` i podrem comprovar el correcte funcionament de LAZY.

Projectes solució:
- `230120_1_BibliotecaV1`
- `230120_2_BibliotecaV1A`
- `230120_3_BibliotecaV1A_Hibernate`
- `230120_4_BibliotecaV1X_Hibernate`

Com controlar les instruccions SQL que executen els proveïdors de JPA	**20/01/23**

- Hibernate:

```
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.max_fetch_depth" value="0"/>
```

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

Hibernate executa les sentències amb consultes preparades amb variables d'enllaç, els valors de les quals no es veuen amb `hibernate.show_sql`. Per visualitzar-les: => No vist a classe, però d'interès per l'alumnat

Informació trobada a https://thoughts-on-java.org/hibernate-logging-guide/

Comprovat funcionament d'apartat JDK Logger (hi ha un error en una propietat)

- Cal retocar el fitxer `logging.properties` que es troba a la carpeta `jre/lib` del JDK que s'utilitzi.
- La propietat `java.util.logging.ConsoleHandler.level` posar-la a `FINEST` (en JDK8 aquesta propietat ja existeix amb valor `INFO`)
- Afegir al final, les propietats:
  ```
  org.hibernate.level=INFO
  org.hibernate.SQL.level=FINEST
  org.hibernate.type.descriptor.sql.level=FINEST
  ```

Evidentment, només activarem això si tenim necessitat de veure els valors que es passa a les variables d'enllaç.

- EclipseLink => No vist a classe, però d'interès per l'alumnat
  - Per veure instruccions SQL:
  ```
  <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"/>
  ```

Generació automàtica de l'identificador	**20/01/23**

Aplicable a camps identificadors enters. Implica que aquest camp passa a ser governat pet JPA.

Per tant, cal fer canvis a les classes on es pugui i vulgui aplicar:

- Retoc del constructor, eliminant el camp identificador.
- Eliminar mètode `setter` corresponent al camp identificador, doncs en cap cas ha de canviar el camp codi.

En l'exemple que estem desenvolupant, classes candidates a tenir generació automàtica de l'identificador: `Soci` i `Prestec`.

4 estratègies: **Tots els projectes exemple que es faciliten tenen un `llegiume` amb explicacions.**

- **AUTO**:
  - 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).

  Exemple: `Projecte 230120_5_ClauAutomàtica_AUTO_Hibernate`

- **IDENTITY**:
  - Comptadors implementats amb camps autonumèrics.
  - Aplicable a SGBD que incorporen camps autonumèrics, com PostgreSQL i MySQL i Oracle 12+

  Exemple: `Projecte 230120_6_ClauAutomatica_IDENTITY_Hibernate`

- **SEQUENCE**:
  - 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 230120_7_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

IdentificacióDeTaula	Comptador
Taula1	23
Taula2	45
...	...

  Exemple: `Projecte 230120_8_ClauAutomatica_TABLE_Hibernate`

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". Els números "reservats" i no "assignats" es perden davant un reinici de l'Oracle. En PostgreSQL les SEQUENCE no tenen aquesta propietat. En Oracle, es pot desactivar el CACHE per a una SEQUENCE, però l'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 dimarts, 24 de gener - Continuació pràctica | **24-01-23**

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`
- `230124_1_BibliotecaV1`
- `230124_2_BibliotecaV1A`
- `230124_3_BibliotecaV1A_Hibernate`
- `230124_4_BibliotecaV1X_Hibernate`

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 P03, 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`, sobreescrigui 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 disjunta	**26/01/23**

Suposem que tenim una classe `Base` (abstracta o no) amb dues classes derivades `Derivada1` i `Derivada2` "disjuntes", 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 `llegiume` amb explicacions.**

- 1 única taula PERSONA per emmagatzemar totes les persones, siguin alumnes o professors. Per tant, contindrà 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: `230126_1A_HerenciaA_SINGLE_TABLE_Hibernate`
  Projecte amb marcatge XML: `230126_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: `230126_2A_HerenciaA_TABLE_PER_CLASS_Hibernate`
Projecte amb marcatge XML: `230126_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 comunies
  - 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: `230126_3A_HerenciaA_JOINED_AmbDiscriminator_Hibernate`
Projecte amb marcatge XML: `230126_3X_HerenciaX_JOINED_AmbDiscriminator_Hibernate`
Projecte amb anotacions: `230126_4A_HerenciaA_JOINED_SenseDiscriminator_Hibernate`
Projecte amb marcatge XML: `230126_4X_HerenciaX_JOINED_SenseDiscriminator_Hibernate`

Hibernate, per gestionar la persistència dels objectes en estratègies `TABLE_PER_CLASS` i `JOINED`, crea unes taules de suport amb prefix `HT`, que no ens han de preocupar (contenen dades temporals).

Exercici - Continuació pràctica	**26/01/23**

Implementar la persistència de les classes `FitxaRevista` i `FitxaLlibre`.

- Tornar a deixar la classe `Fitxa` abstracta, amb mètode `toString` abstracte.
- Utilitzar l'estratègia `JOINED` que és la que estem més acostumats a utilitzar.
- Comprovar funcionament en Oracle-Hibernate.

Solució:

- Via anotacions, incorporem anotacions vinculades a estratègia `JOINED` en classe `Fitxa` (que hem fet abstracta) i en classes `FitxaRevista` i `FitxaLlibre`.
- Via marcatge XML, incorporem anotacions vinculades a estratègia JOINED 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`.
- Taula `F_LLIBRE` per enregistrar les dades específiques dels objectes `Llibre`.

Projectes solució:
- `230127_1_BibliotecaV1`
- `230127_2_BibliotecaV1A`
- `230127_3_BibliotecaV1A_Hibernate`
- `230127_4_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 i les modifica.
  Comprovar, després d'executar-lo, la informació que hi ha a les 3 taules

Herència no disjunta	**27/01/23**

JPA no disposa d'un mecanisme per aquesta situació.
Un exemple aquí on es proposa que si es vol tenir les dades comunes en una taula i les dades específiques en altres taules, es munti una relació entre la classe base i les classes derivades amb un identificador derivat.

Herència amb classe base no persistent	**27/01/23**

Usar `@MappedSuperclass/mapped-superclass`. Els camps de la classe base s'hereten com a columnes en les classes derivades. No existeixen entitats de la classe base i no es pot cercar (`find-JPQL`) per aquesta 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.

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

Problemes:
- 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`.
- El programa P01 no elimina la taula FITXA existent. Cal fer `drop` previ de manera manual.
- El programa P03 no pot fer un `find(Fitxa.class...` Cal saber si es va a cercar un `FitxaRevista` o un `FitxaLlibre`.

Projectes solució (eliminant el marcatge de la classe `Prestec`):
- `230127_5_BibliotecaV1`
- `230127_6_BibliotecaV1A`
- `230127_7_BibliotecaV1A_Hibernate`
- `230127_8_BibliotecaV1X_Hibernate`

Objectes incrustats (classes `@Embeddable/embeddable`)	**31/01/23**

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 `CorreuElectrònic`, 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: `230131_A_ObjecteIncrustatA_Hibernate`
Projecte amb marcatge XML: `230131_A_ObjecteIncrustatX_Hibernate`

Els projectes contenen un `llegiume` amb informació.

Col·leccions d'objectes bàsics (classes facilitades per Java)	**31/01/23**

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 de la FK
  - `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: `230131_B_ColleccioObjectesBasicsA_Hibernate`
Projecte amb marcatge XML: `230131_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: `230131_C_ColleccioObjectesBasicsAmbOrdreA_Hibernate`
Projecte amb marcatge XML: `230131_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`)	**31/01/23**

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:

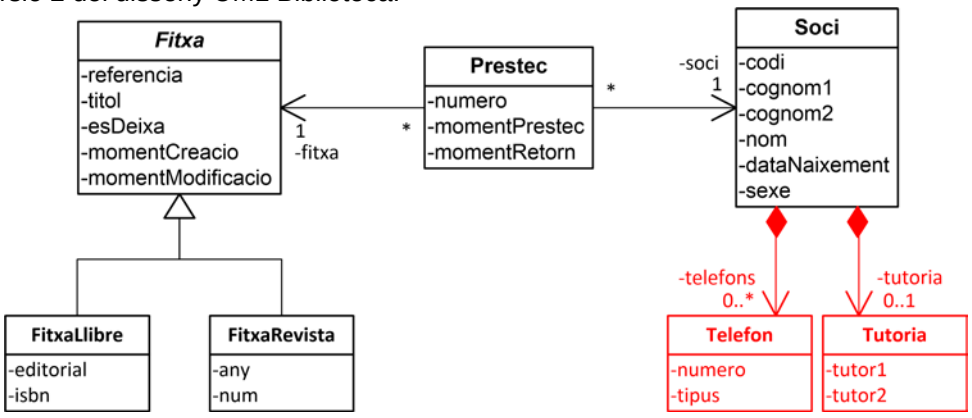
- `ObjecteIncrustat`
- `ColleccioObjectesBàsics`

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

Projecte amb anotacions: `230131_D_ColleccioObjectesIncrustatsA_Hibernate`
Projecte amb marcatge XML: `230131_D_ColleccioObjectesIncrustatsX_Hibernate`

Exercici per dijous, 2 de febrer - Continuació pràctica | **02/02/23**

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!!!
- 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 `230202_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 `230202_2_BibliotecaV2A`

- 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 `230202_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 `230202_4_BibliotecaV2X_Hibernate`

Claus compostes constituïdes per camps bàsics — 02/02/23

Com actuar davant una clau composta? Ho exemplificarem 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: `230203_1A_ClauCompostaViaMultiplesIdA_Hibernate`
Projecte amb marcatge XML: `230203_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 `EntityManger.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`**

  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="nompaquet.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: `230203_2A_ClauCompostaViaIdClassA_Hibernate`
Projecte amb marcatge XML: `230203_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:
    - No ha de contenir marques `@Id/id` (les classes inscrustades no en poden tenir)
    - 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: `230203_3A_ClauCompostaViaEmbeddedIdA_Hibernate`
  Projecte amb marcatge XML: `230203_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`	**02/02/23**

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 `230203_4A_ClauCompostaAmbMany2OneViaIdClassA-V01_Hibernate`
- Amb `EmbeddedId`:
  Projecte `230203_5A_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 `230203_6A_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 `230203_7A_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.

Versió que permet utilització "fàcil" de mètode `find` i vàlida per *Hibernate* i *EclipseLink*:

- Retocs a efectuar a la classe que conté la clau composta – versió `@IdClass`
  - 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 `llegiume` 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 `230203_8A_ClauCompostaAmbMany2OneViaIdClassA-V02_Hibernate`
    Projecte `230203_8A_ClauCompostaAmbMany2OneViaIdClassA-V02_EclipseLink`

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

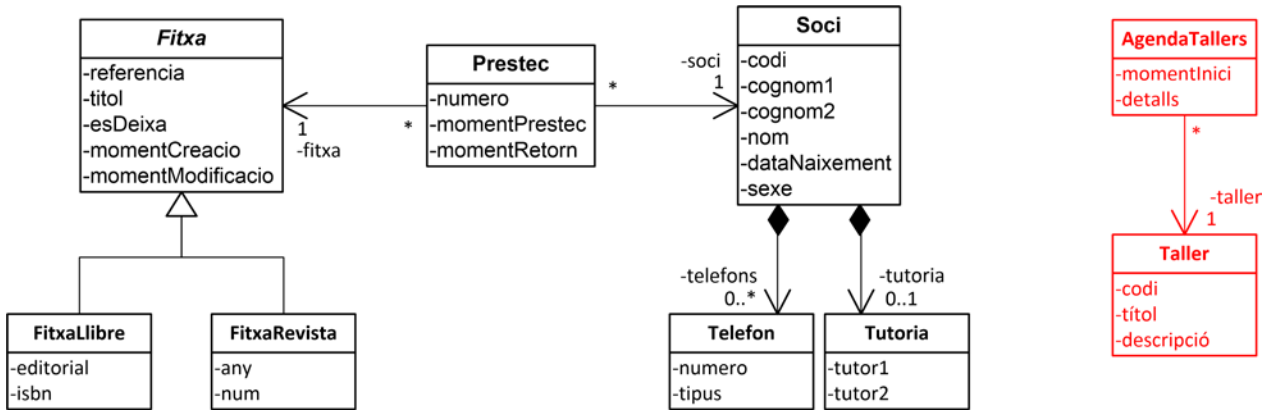
- Retocs a efectuar a la classe que conté la clau composta – versió `@EmbeddedId`
  - El camp `ManyToOne` o `OneToOne` que forma part de la clau composta, mantenir-lo a la classe original (no a la classe incrustada) amb marcatge `ManyToOne` o `OneToOne` però sense `@Id/id` i incorporar dins `@JoinColumn` 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 `230203_9A_ClauCompostaAmbMany2OneViaEmbeddedIdA-V02_Hibernate`
Projecte `230203_9A_ClauCompostaAmbMany2OneViaEmbeddedIdA-V02_EclipseLink`

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

Exercici per dimarts, 7 de febrer - Continuació pràctica | **03/02/23**

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, 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

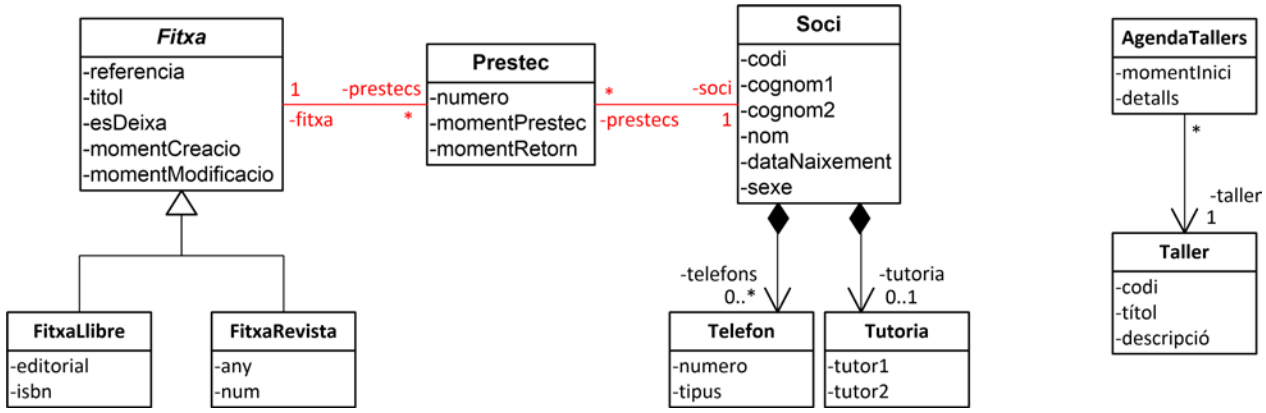
Solució utilitzant `@IdClass` per la clau composta de la classe `AgendaTallers`:

- Projecte `230207_1_BibliotecaV3`
- Projecte `230207_2_BibliotecaV3A`
- Projecte `230207_3_BibliotecaV3A_Hibernate`
- Projecte `230207_4_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`	**09/02/23**

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).
- `prestecs` 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). **Això també es veu 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:
    - 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`.
    - 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 que el `prestec` canviés 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:

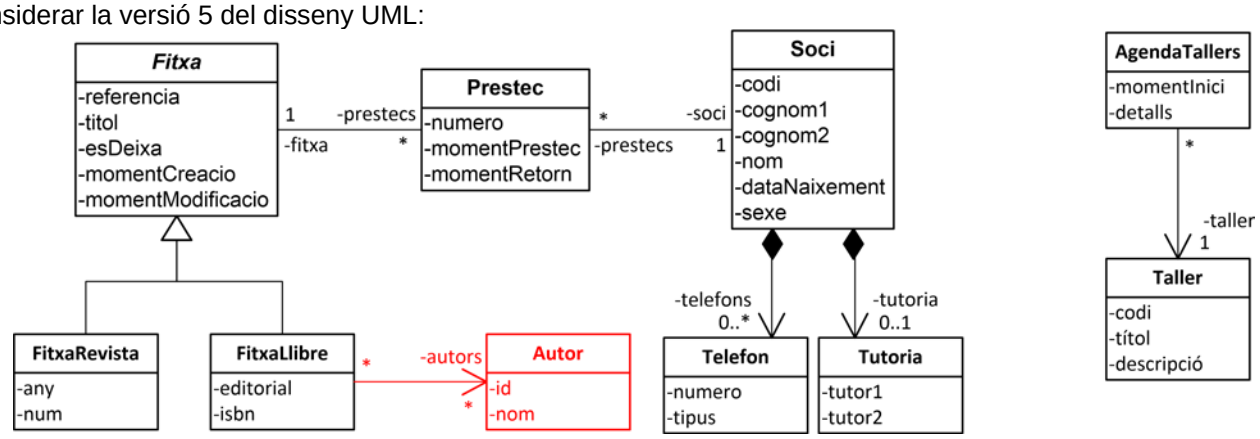
- `230209_1_BibliotecaV4`
- `230209_2_BibliotecaV4A`
- `230209_3_BibliotecaV4A_Hibernate`
- `230209_4_BibliotecaV4X_Hibernate`

Comentaris:

- En el projecte de proves s'ha afegit una classe `Utils` amb un mètode que s'invoca a P2 i a P3 i que es pot usar quan es vulgui, que en posar en marxa el programa pregunta a l'usuari si vol veure les instruccions SQL que executa *Hibernate* sense necessitat de modificar la propietat `hibernate.show_sql` en el fitxer `persistence.xml`.
- En els programes P2 i P3, si es respon S a la pregunta de mostrar instruccions SQL, el programa dona per pantalla explicacions sobre les instruccions que executa *Hibernate*. Els constructors i algun mètode `setter` conté `sout` per poder comprovar quan s'executa.

Continuació pràctica – Relacions `ManyToMany` unidireccional (`Autoria` de `Llibre` a `Autor`)	**09/02/23**

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 podrem 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:

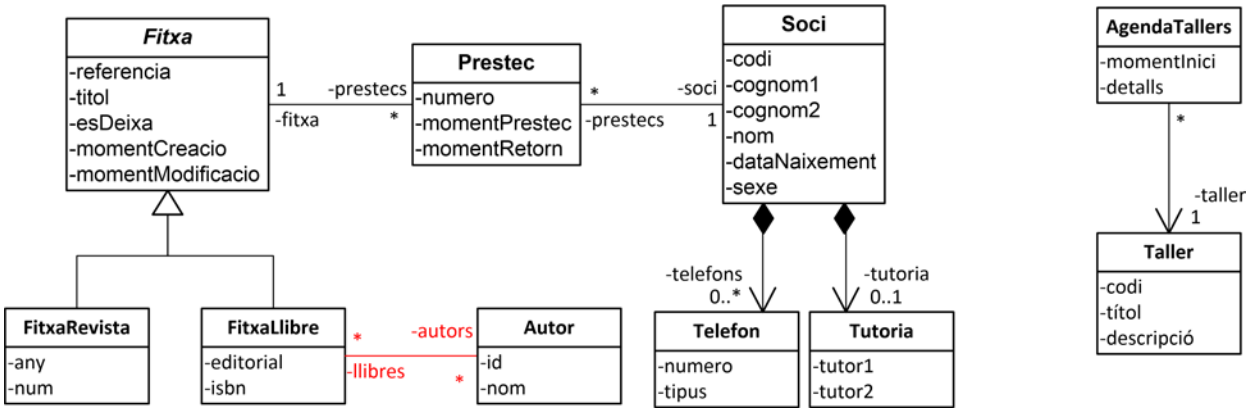
- `230209_5_BibliotecaV5`
- `230209_6_BibliotecaV5A`
- `230209_7_BibliotecaV5A_Hibernate`
- `230209_8_BibliotecaV5X_Hibernate`

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`.
- 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`.

Continuació pràctica – Relacions `ManyToMany` bidireccional (`Autoria` entre `Llibre` i `Autor`)	**09/02/23**

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` (però tota la definició de la taula ja és a l'altra classe)

Projectes:

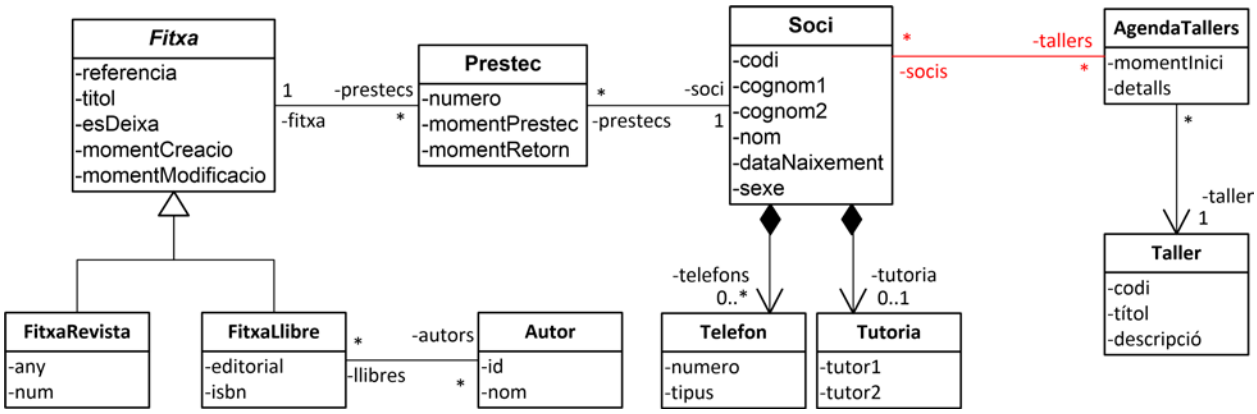
- `230209_A_BibliotecaV6`
- `230209_B_BibliotecaV6A`
- `230209_C_BibliotecaV6A_Hibernate`
- `230209_D_BibliotecaV6X_Hibernate`

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 divendres, 10 de febrer - Continuació pràctica – Pràctica de relacions ManyToMany	**09/02/23**

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:

- `230210_1_BibliotecaV7`
- `230210_2_BibliotecaV7A`
- `230210_3_BibliotecaV7A_Hibernate`
- `230210__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`	**10/02/23**

a) `em.detach(obj)`
Allibera un objecte de l'EntityManager, de manera que aquest ja no el controla.

b) `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`.

c) `em.refresh(obj)`
Refresca un objecte gestionat per l'EntityManager amb les dades existents a la BD (per si algún altre procés hagués efectuat canvis a l'objecte a la BD)

d) `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: `230210_5_BibliotecaV7A_Hibernate_ProvesPersistenciaObjectes`

- Programa P02 amb proves de `merge`
- Programa P03 amb proves de `refresh` i `remove`

Esdeveniments (triggers) en JPA	**14/02/23**

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_2_2.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_2_2.xsd`.

Consultes en JPQL: consultes JPA i consultes natives SQL	**16/02/23**

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:
- `230216_1_BibliotecaV7`
- `230216_2_BibliotecaV7A`
- `230216_3_BibliotecaV7A_Hibernate`
- `230216_4_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	**16/02/23**

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.

Projecte amb exemples:
- `230216_5_BibliotecaV7A_Hibernate_ProvesInstruccionsNoSelect`

- 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 que hi ha les 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_SOCI) - s'ha trobat un registre fill

  Resum:
  - En intentar eliminar un `Soci` que té agendes, JPA executa `delete` previ de les agendes.
  - 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 Odoo 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 `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 que hi ha les 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.
  Mireu els comentaris que conté.

- Programa P06: Comprovació que l'`EntityManager` no s'assabenta de les modificacions/eliminacions dels objectes que està gestionant.
  El programa mostra missatges explicatius del què està succeint en cada moment.
  Observeu que s'utilitza `refresh` per refrescar canvis a la BD i `detach` per eliminar objectes de l'EM.

Utilització de `CascadeType.REMOVE` i/o FK amb `ON DELETE CASCADE`	**17/02/23**

En el darrer projecte, el programa P03 ens ha mostrat que si no hi ha definida l'eliminació en cascada a nivell de JPA (`CascadeType.REMOVE`) o a nivell de FK a la BD (on `delete cascade`), el programador ho ha de tenir en compte i fer les eliminacions prèvies que correspongui.

En els marcatges efectuats fins la V7 de `Biblioteca`, no hem usat enlloc:

- `CascadeType.REMOVE` per a que sigui JPA qui actuï amb eliminació prèvia.
- `On delete cascade`, en crear la BD, definició que habitualment s'efectuarà via guió.

En cas que s'encarregui JPA de crear la BD, per aconseguir una FK amb eliminació en cascada, cal usar l'atribut `foreignKeyDefinition`/`foreign-key-definition` dins `@ForeignKey`/`foreign-key`.

Per a practicar-ho, decidim retocar el model Biblioteca a V8, amb els retocs:

- Definir `CascadeType.REMOVE` en les relacions:
  - `OneToMany Soci.prestecs`
    També podríem definir, via FK, on `delete cascade` en la `JoinColumn` de `Prestec.soci`.
  - `OneToMany Fitxa.prestecs`
    També podríem definir, via FK, on `delete cascade` en la `JoinColumn` de `Prestec.fitxa`.
- Definir on `delete cascade` en la FK de la `JoinColum`:
  - De la relació `AgendaTallers.taller`
    No es pot usar `CascadeType.REMOVE`, doncs a `Taller` no hi ha relació a `AgendaTallers`.
  - De la col·lecció `Soci.telefons`
    No existeix la clàusula `cascade` en els `ElementCollection`.
  - En les FK de la `JoinTable` `inscripcio` dins `AgendaTallers.socis`
  - En les FK de la `JoinTable` `autoria` dins `FitxaLlibre.autors`

Projectes amb els retocs incorporats:

- `230217_1_BibliotecaV8`
- `230217_2_BibliotecaV8A`
- `230217_3_BibliotecaV8A_Hibernate`
- `230217_4_BibliotecaV8X_Hibernate`

El programa P03 demostra quan JPA aplica `CascadeType.REMOVE` i quan és el SGBD que aplica on `delete cascade` definida en FK. Recordar que :

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

Tipus d'accès de JPA a les dades: `FIELD` – `PROPERTY`	**21/02/23**

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 interessi.

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 sobreescriure 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 duplicitat 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 interessi 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:

- `230221_1_BibliotecaV9`
- `230222_2_BibliotecaV9A`
- `230223_3_BibliotecaV9A_Hibernate`
- `230224_4_BibliotecaV9X_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.
- I com s'emplena? A partir de quin càlcul? 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 haurem 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.

Enunciat examen curs 2017-2018

Material en el projecte NetBeans `RRHH` facilitat.

Considereu el disseny MR de la dreta, pensat per gestionar informació vinculada als recursos humans d'una empresa.

Creeu en el SGBD Oracle un esquema de nom `RRHHV1` on executar el guió (facilitat dins el projecte) `RRHH_Oracle_Schema&Data.sql` , que conté la definició de taules per l'Oracle i joc de proves.

Totalment prohibit efectuar cap modificació en aquest esquema!!!

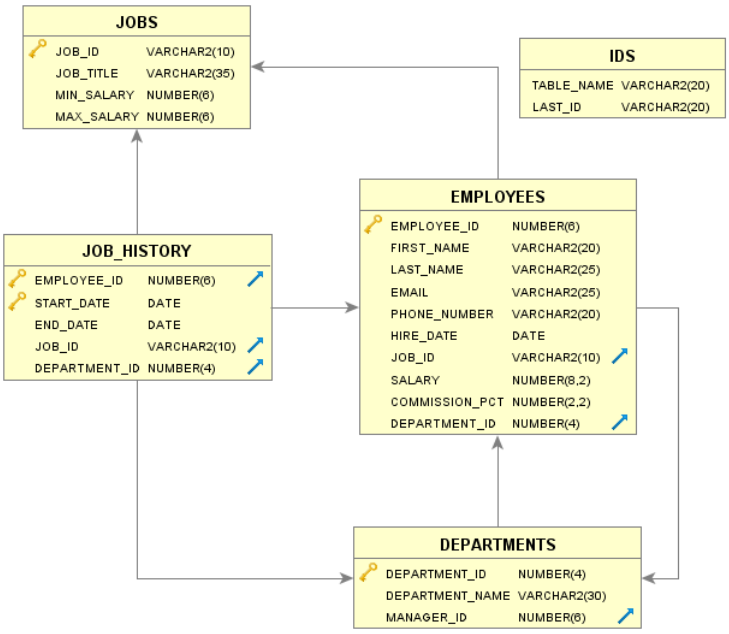
La taula `IDS` està pensada per contenir comptadors dels identificadors automàtics de diverses taules. En aquest moment, però, només conté el comptador per la taula `EMPLOYEES` com podeu comprovar si feu una consulta del seu contingut.

Els informàtics d'aquesta empresa han dissenyat unes classes Java (esquema UML de la dreta) per gestionar les dades de la BD, i volen aprofitar les facilitats que dona JPA (concretament EclipseLink i Hibernate) però no tenen experiència, ni en JPA ni en el disseny de classes amb relacions bidireccionals i necessiten de la vostra ajuda.

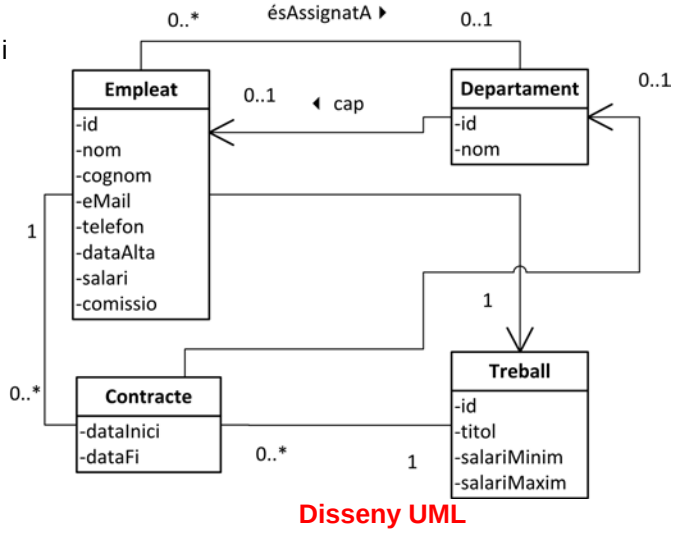
Es demana aplicar un seguit de millores i/o retocs al projecte `RRHH` facilitat i desenvolupar nous projectes.

Les classes estan "pelades"; només contenen la definició dels camps...

No perdeu temps creant mètodes i més mètodes. Només poseu els mètodes que siguin necessaris i/o imprescindibles pel què es demana.



Disseny MR



Disseny UML

Exercici 1:

Reanomenar el projecte (i carpeta) `RRHH` facilitat per `RRHHV1` on cal retocar i marcar adequadament les classes `Empleat` i `Treball` via XML i les classes `Contracte` i `Departament` via anotacions per a que JPA pugui gestionar adequadament el mapatge entre les classes i l'esquema `RRHHV1` generat amb el guió facilitat, tenint en compte que:

- L'eliminació d'un empleat ha de suposar l'eliminació automàtica dels seus contractes.
- L'eliminació d'un departament no ha de suposar l'eliminació dels seus empleats, els quals quedaran sense departament assignat.
- La persistència d'un departament ha de provocar la persistència dels empleats encara no persistents que pugui tenir assignats.
- Cal implementar la correcta sincronització de les relacions bidireccionals.
- La classe `Empleat` ha d'incorporar les consultes JPQL amb nom següents:
  - `trobaTots`, que recuperi tots els empleats
  - `trobaEmpleatPerId`, que recuperi l'empleat amb identificador passat per paràmetre
  - `trobaContractesDeEmpleatPerId`, que recuperi els contractes d'un empleat amb identificador passat per paràmetre

Crear projectes de nom `RRHHV1Hibernate` i `RRHHV1EclipseLink` per usar el marcatge definit a `RRHHV1` via `Hibernate` i via `EclipseLink` contra l'esquema `RRHHV1` generat pel guió `sql` facilitat, de manera que `Hibernate` ni `EclipseLink` efectuïn cap modificació a l'esquema `RRHHV1` de la BD.

Cada projecte ha d'incorporar les biblioteques imprescindibles (només les imprescindibles).

Per comprovar la coherència del marcatge amb la BD, incorporeu a cada projecte (`Hibernate` i `EclipseLink`) el programa `P01_ComprovaEsquema` que validi el marcatge però no modifiqui l'esquema `RRHHV1`.

Exercici 2:

Incorporeu en el projecte `RRHHV1Hibernate`, els següents programes:

- `P02_ConsultaEmpleats`, que invoqui la consulta `trobaTots` i mostri les dades de cada empleat, invocant mètode `toString` (generat per NetBeans) de classe `Empleat`.
- `P03_ConsultaContractesEmpleat`, que demani la introducció d'un codi d'empleat per consola i:
  - Comprovi la seva existència (via `trobaEmpleatPerId`), mostrant-lo (`toString`) o informant de la no existència
  - Si hi és, mostri els seus contractes, via `trobaContractesDeEmpleatPerId`, mostrant-los un a un, invocant mètode `toString` (generat per NetBeans) de classe `Contracte`.
- `P04_GestióDades1` que afegeixi a la BD un departament i un empleat assignat al departament i en sigui el seu cap.
- `P05_GestióDades2` que demani per consola un codi de treball i un percentatge d'augment estrictament positiu i procedeixi a aplicar el percentatge als salaris mínim i màxim del treball indicat així com a tots els empleats que en aquest moment tenen assignat aquest treball. Cal controlar l'existència del treball.

Exercici 3:

Creeu en Oracle un nou esquema buit de nom `RRHHV2`.
Feu còpia del projecte `RRHHV1` desenvolupat en exercici1 amb nom `RRHHV2`.
Feu còpia del projecte `RRHHV1EclipseLink` a `RRHHV2EclipseLink` per usar el mapatge de `RRHHV2`.

Retocar els dos projectes V2 per aconseguir que l'execució del programa `P01` de `RRHHV2EclipseLink` creï les taules dins l'esquema `RRHHV2` calcades a la creació que n'efectua el guió `sql` facilitat.

Passos seguits per aconseguir la solució – Projectes RRHHV1... i RRHHV2...

- Completar les classes (els mètodes `setter` estan incomplets, caldria afegir les comprovacions)
  - Serializables
  - Donat que a l'exercici 2-P04 hem de procedir a crear objectes de diverses classes (`Departament`, `Empleat` i `Treball`), necessitem dotar com a mínim a aquestes classes d'un constructor. Cap problema si es proporciona constructor per a totes les classes.
  - Classe `Empleat`:, donat que la clau (camp `id`) és autonumèrica:
    - El constructor no pot contenir paràmetre `id`.
    - No pot tenir mètode `setId`.
  - Constructor sense paràmetres `protected` per a totes les classes que tenen altre constructor.

- Següents mètodes `setter` privats per què el(s) camp(s) identificadors han de ser immutables:
  - `Departament.setId`
  - `Treball.setId`
  - `Contracte.setEmpleat`
  - `Contracte.setDataInici`
- Crear el fitxer `persistence.xml` adequat.
  - Posarem dues UP, una per Hibernate i l'altra per EclipseLink, ja que ens demanen provar-ho amb els dos proveïdors.
  - Incorporarem les dades de connexió que corresponguin.
  - Incorporarem en elements `class` totes les classes gestionades per JPA
  - Incorporarem en elements `mapping` els fitxers XML que continguin marcatge XML d'algunes classes. Recordeu que un fitxer XML pot contenir marcatge de vàries classes
- Sincronització de relacions bidireccionals:
  - Relació `ésAssignatA` entre `Empleat` i `Departament`
    - `Empleat.setDepartament`, que ha de permetre departament nul.
    - `Departament.iteEmpleats, Departament.addEmpleat` i `Departament.removeEmpleat`.
  - Relació entre `Contracte` i `Empleat`.
    - `Contracte.setEmpleat`, que NO ha de permetre empleat nul.
    - `Empleat.iteContractes` i `Empleat.addContracte`.
  - Relació entre `Contracte` i `Treball`.
    - `Contracte.setTreball`, que NO ha de permetre treball nul.
    - `Treball.iteContractes` i `Treball.addContracte`.
- Marcatge via anotacions/XML segons requeriments:
  - En RRHHV1, només introduim el marcatge per a que JPA pugui sincronitzar les classes amb les taules i es garanteixin els requeriments de funcionament indicats.
    Per tant, introduim:
    - Marca per indicar nom de taula si no coincideix amb el nom de la classe
    - Marca per indicar nom de columna quan no coincideixi amb el nom del camp dins la classe
    - Marca per indicar la clau
    - Marques per la clau automàtica a `Empleat`.
    - Marques i muntatge per aconseguir la clau automàtica composta a `Contracte` on una part de la clau és camp `many2one`. Es pot qualsevol dels 2 mecanismes aplicats en classe `AgendaTallers` :
      Via `@IdClass` com en projectes `210408_7_ClauCompostaAmbMany2OneViaIdClass...`
      Via `@EmbeddedId` com en projectes `210408_8_ClauCompostaAmbMany2OneViaEmbeddedId...`
    - Marca `@ManyToOne/many-to-one` per les relacions molts a un.
      Introduir `Fetch.Lazy` (no s'explicitava però millor fer-ho)
      Introduir `Cascade.persist` (no s'explicitava però millor fer-ho)
    - Informar de tots els camps que no són obligatoris.
    - Marca `@OneToMany/one-to-many` per les relacions un a molts que han de tenir inversa
    - Les consultes amb nom indicades a la classe `Empleat`.
    - En principi no afegim cap clau forana doncs ja tenim la BD creada.
    - Respecte: *L'eliminació d'un empleat ha de suposar l'eliminació automàtica dels seus contractes*.
      Cal afegir `cascade-remove` a la relació `one-to-many` d'`Empleat` cap a `Contracte`.
    - Respecte: *La persistència d'un departament ha de provocar la persistència dels empleats encara no persistents que pugui tenir assignats.*
      Cal afegir `CascadeType.PERSIST` a la relació `OneToMany` de `Departament` cap `Empleat`
    - Respecte: *L'eliminació d'un departament no ha de suposar l'eliminació dels seus empleats, els quals quedaran sense departament assignat.*
      La clàusula `cascade` no permet definir un comportament similar a `on delete set null` de la definició de les claus foranes i en aquesta BD no tenim la clau forana `EMP_DEPT_FK` amb on `delete set null`.
      La solució és usar la funcionalitat de triggers que ens facilita JPA, vista aquí.
      Per tant, incorporem a `Departament` el següent contingut:
      `@PreRemove`

```
    private void posarNullEnEmpleats() {
        empleats.forEach(e -> e.setDepartament(null));
    }
```

El codi anterior és Java-ERRONI (excepció que no té a veure amb JPA). El projecte RRHHV1 conté programa `ProvaEliminacióDepartamentEmpleats` que serveix per comprovar l'eliminació de departament 10 (que conté 1 empleat) i departament 30 (que conté 6 empleats). Si executem el programa amb la versió `posarNullEnEmpleat` anterior, cap problema en eliminar departament 10 però en eliminar departament 30 apareix `java.util.ConcurrentModificationException` deguda a que s'està intentant modificar una col·lecció en un procés iteratiu i això Java no ho permet. Funciona quan la llista `empleats` de departament només conté 1 element, però quan n'hi ha més, es produeix l'excepció indicada.

Possible solució: Volcar els empleats de la llista en una taula i iterar sobre la taula. És a dir:

```
    @PreRemove
    private void posarNullEnEmpleats() {
        Object t[] = empleats.toArray();
        for (Object o: t) {
            ((Empleat)o).setDepartament(null);
        }
    }
```

- En RRHHV2, a partir de la còpia de RRHHV1, afegim marcatge necessari per intentar crear la BD el més similar possible a la creació que en fa el guió facilitat:
  - Afegim la definició de les claus foranes.
  - Afegim restricció d'unicitat de correu electrònic a la taula `EMPLOYEES`.

- Programes demanats:
  - Sobre RRHHV1:
    - En les dues UP posem `javax.persistence.schema-generation.database.action` a none. En Hibernate, si s'usa la seva propietat `hibernate.hbm2ddl.auto` a `validate`, per bug d'Hibernate, dona missatges d'error en validar el nom de les FK via marcatge XML. En EclipseLink no es disposa d'opció de validació.
    - P01 crea l'EntityManager, en Hibernate i en EclipseLink indicant la UP que correspon
    - P02-P03-P04-P05 segons requeriments. Res especial a observar.
  - Sobre RRHHV2:
    - Posem `javax.persistence.schema-generation.database.action` a `create-or-extend-tables` a partir del requeriment que cal crear l'estructura en una BD buida.