

Frameworks de persistencia

Tabla de contenidos

- 1) Problema inicial
- 2) Patrones en la arquitectura de acceso a datos
- 3) Caso de estudio: patrón de diseño DAO
- 4) Solución ORM

Problema inicial

- Persistencia de datos como concepto fundamental
 - Persistir: *"Durar por largo tiempo"*. RAE
- ¿Admitiríamos un sistema de información que pierde todos los datos de un día para otro?

Persistencia

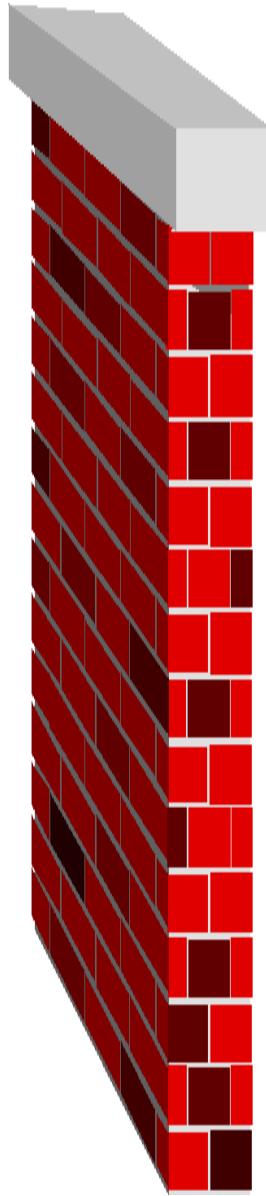
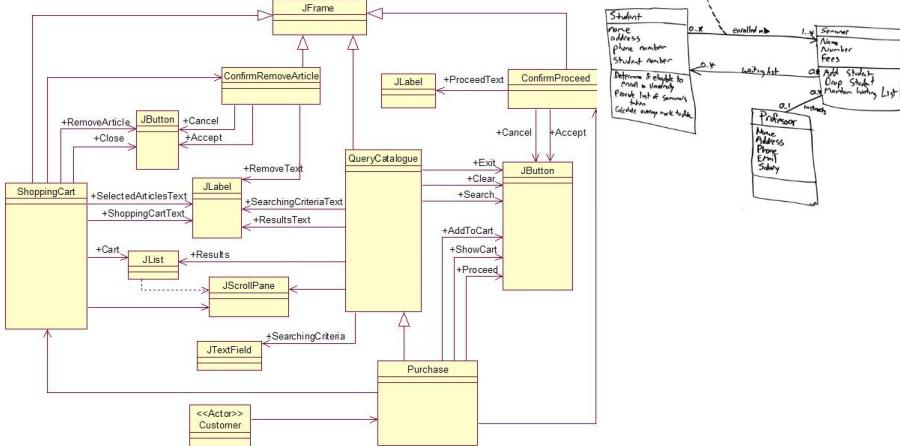
- Almacenar datos en una base de datos relacional como solución habitual
- Aunque también hay otras opciones:
 - Almacenar datos en ficheros planos
 - Almacenar XML, JSON, etc. (Ojo, también en bases de datos...)
 - Serializar objetos (mecanismos muy ligados al lenguaje de programación)

Ej: “nuestra aplicación”

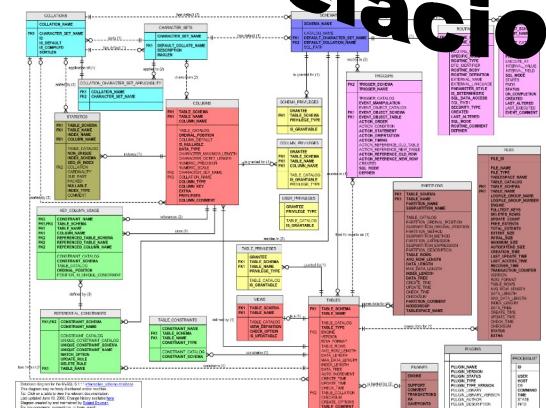
Problema inicial



Desarrollo de Prog. Orientado a aplicaciones



Bases de datos Modelo relacional



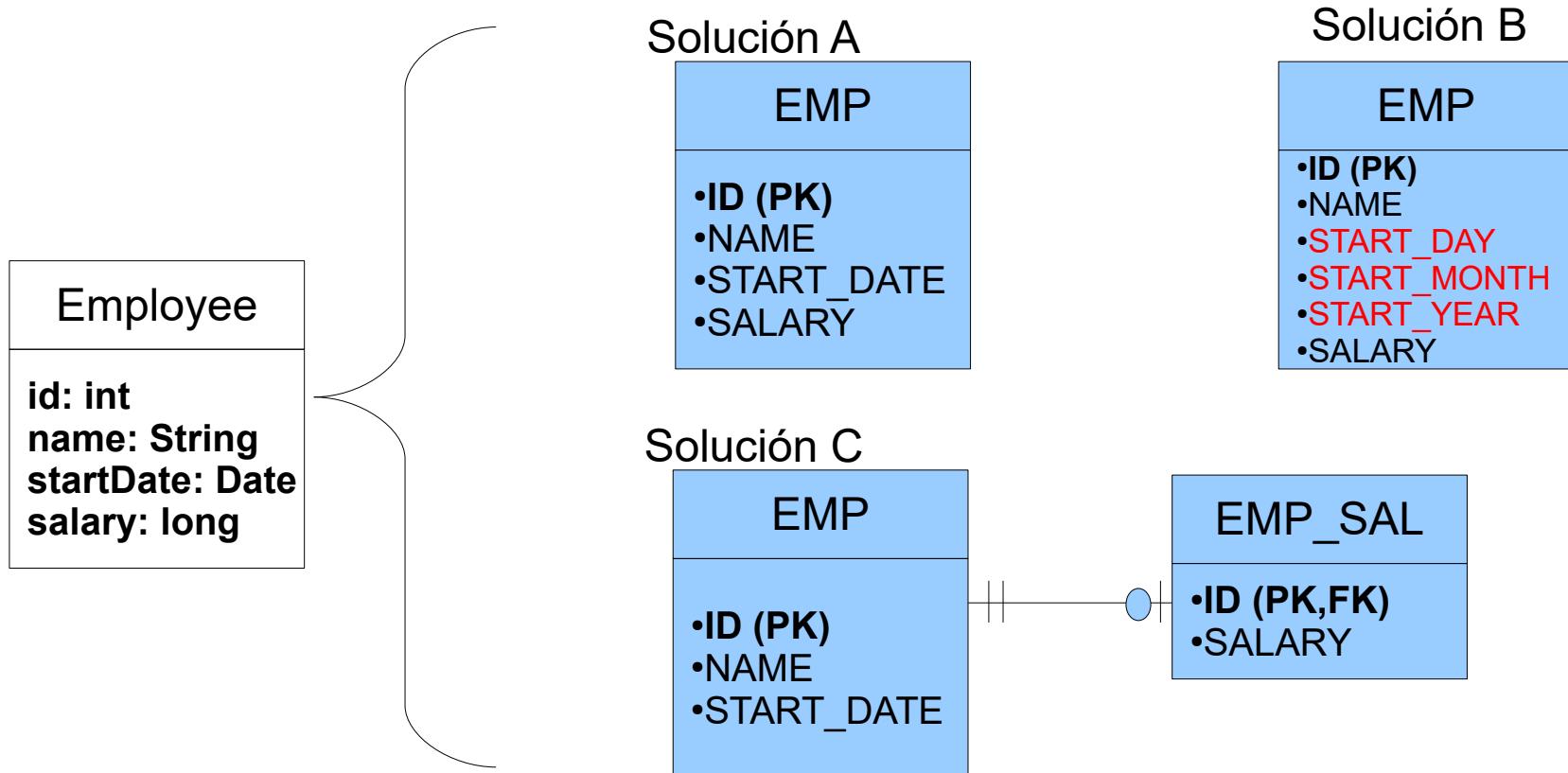
Problema inicial

- Vital el desarrollo de aplicaciones que **interactúen** con la BD (no restringido a LPOO)
 - Ej: Oracle (Sun) con Java y drivers JDBC
 - Ej: Microsoft con ODBC, OLEDB, ADO.NET, ADO.NET Entity Framework, ADO.NET Data Services, WCF Data Services, etc.
 - Utilizando los lenguajes de programación más extendidos... que ahora mismo son **estructurados** (Ej: C) u **orientados a objetos (o híbridos)** (Ej: Java, Objective-C, C++, C#, Python, PHP, etc.)
- **Problema de partida:**
 - ¿Existe **equivalencia/simetría** entre el concepto de **clase (POO)** y **tabla (BD Relacional)**?
 - ¿Entre **objetos** y **registros**?

Problema inicial

Impedance Mismatch

- Ej: Clase vs. Tablas

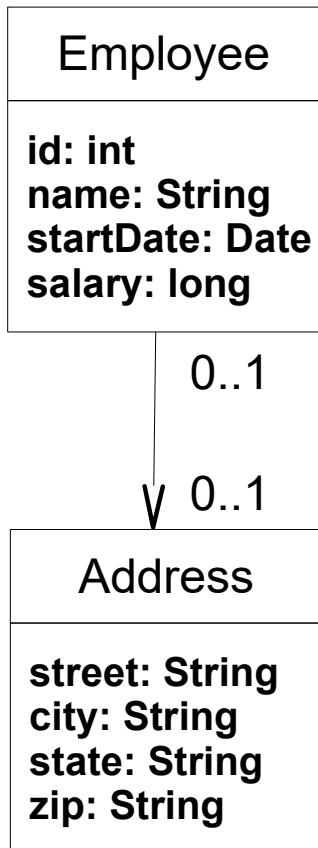


- **Las necesidades de la BD ganan a la aplicación:**
 - Podría haber diferentes tipos de aplicaciones contra la misma BD
- **Las aplicaciones deben ajustarse el modelo de la BD**

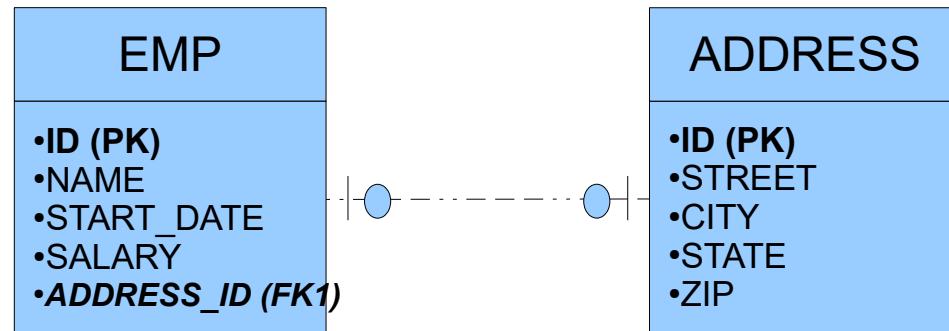
Problema inicial

Impedance Mismatch

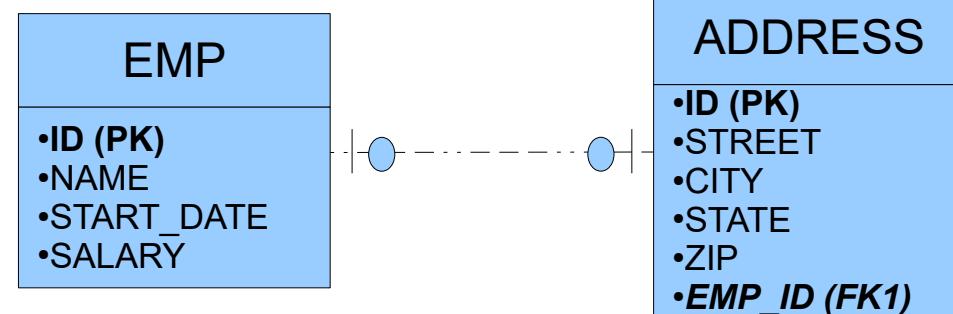
- Ej: Relaciones



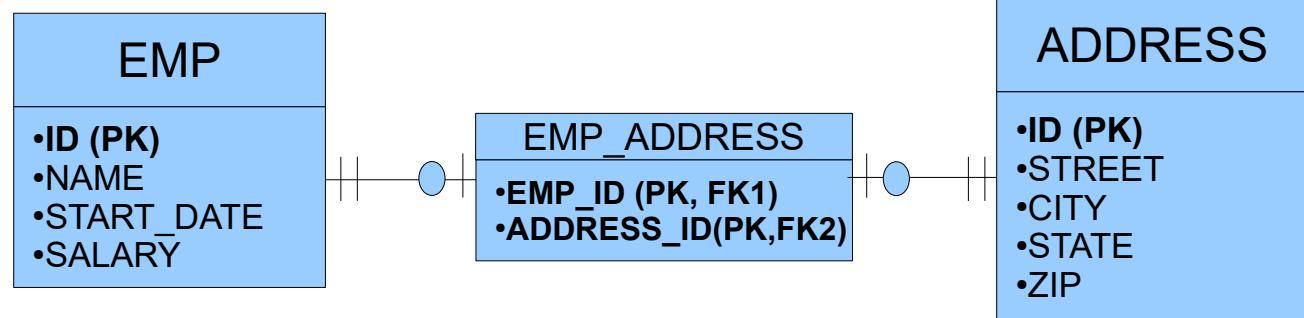
Solución A



Solución B



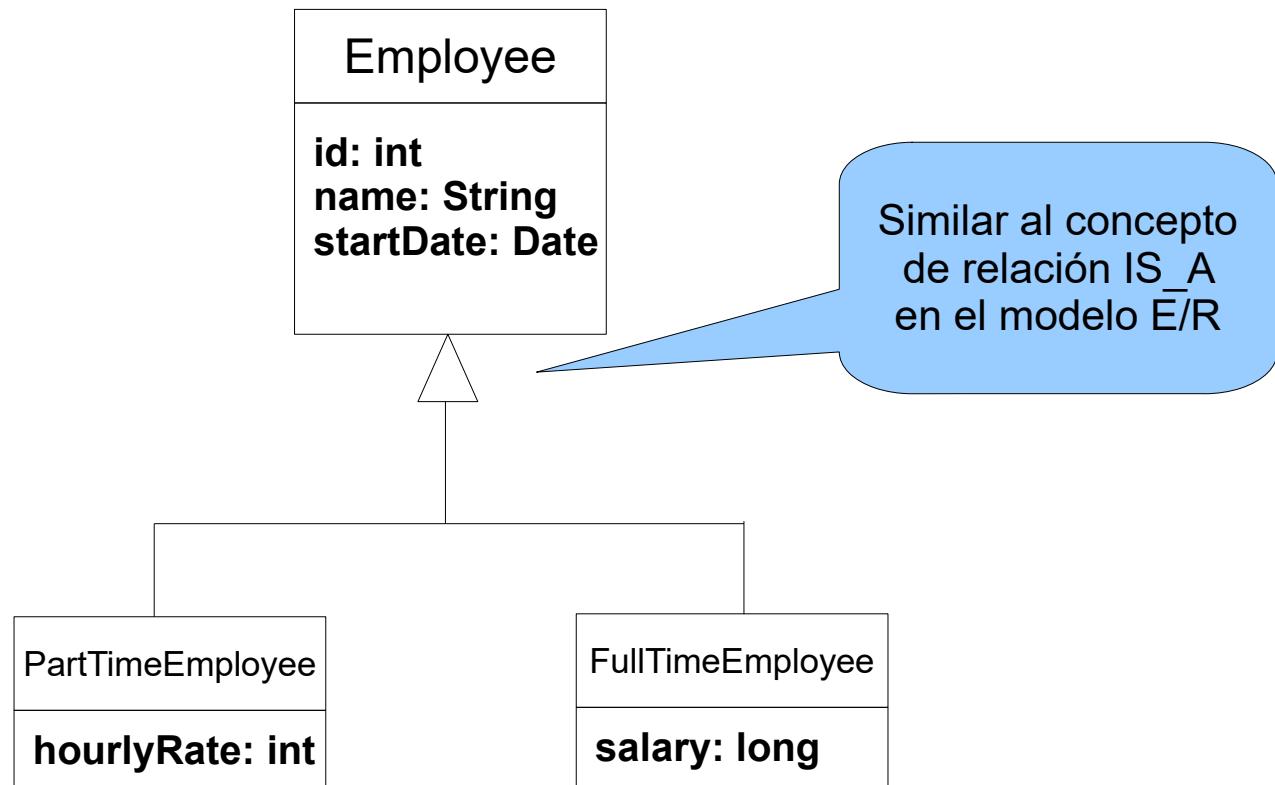
Solución C



Problema inicial

Impedance Mismatch

- Ej: Herencia

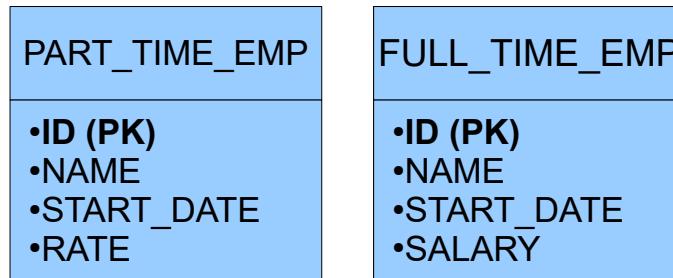


Problema inicial

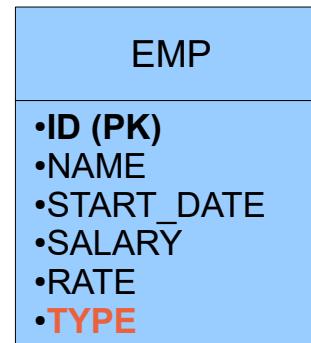
Impedance Mismatch

- Ej: Herencia

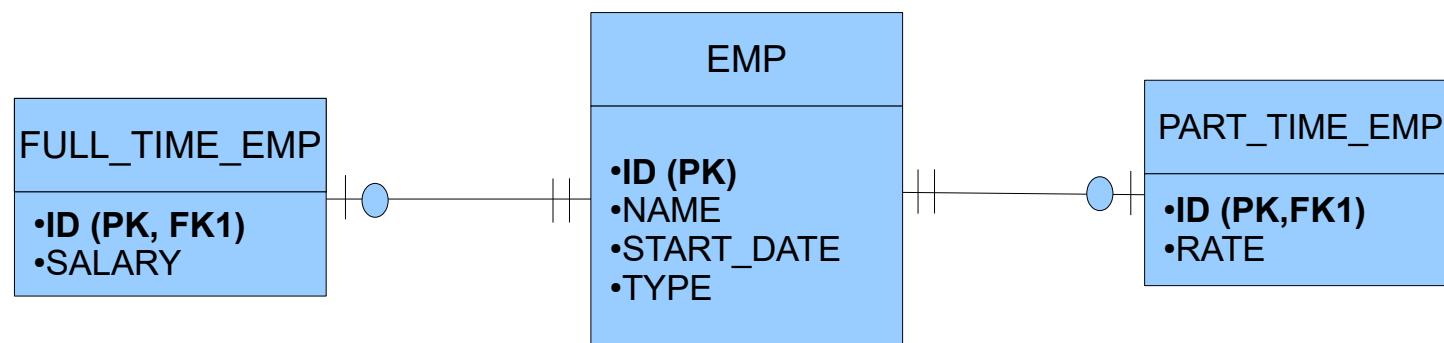
Solución A



Solución B



Solución C



Problema inicial

Soluciones intermedias en desarrollo

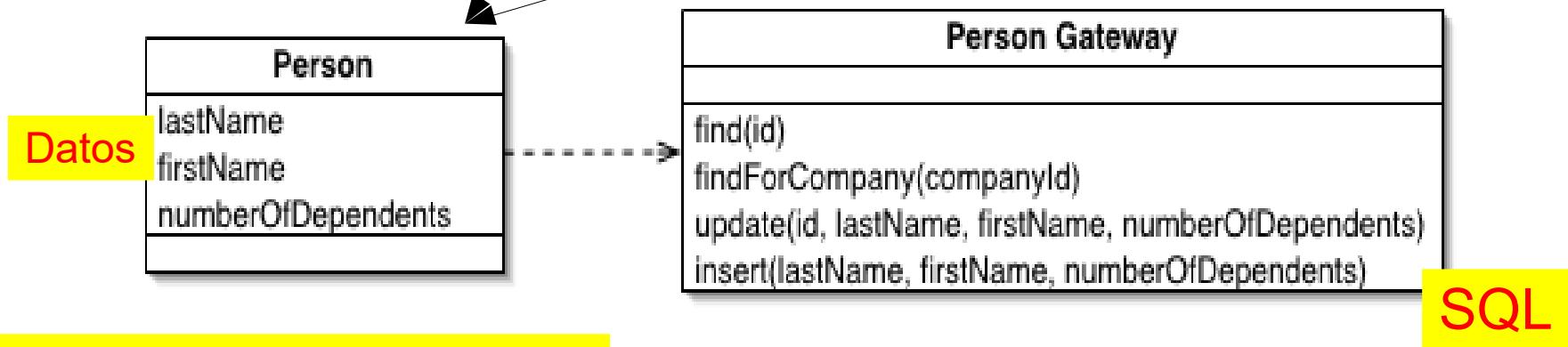
- JDBC embebido a lo largo de TODA la aplicación
 - ¡Alto acoplamiento de la capa de persistencia con la aplicación!
 - ¿Qué ocurre si cambiamos de producto (SGBD)?
 - ¿Qué ocurre si dejamos de usar un SGBD? (Recordemos que hay otras soluciones...)
 - **Impedancia entre ResultSet (o RowSet) y clases/objetos**
- Solución para desacoplar en ingeniería del software:
 - Patrones en la arquitectura de la aplicación
 - Patrón de diseño
 - Ej: DAO
 - Pero exigen un alto esfuerzo (y experiencia) de diseño e implementación
- Modelos de componentes (distribuidos) Ej: Enterprise JavaBeans y EntityBeans, DCOM, COM+, .NET, etc.
 - Excesivamente complejos
- Solución final: *frameworks* de persistencia (Mapeo Objeto-Relacional, Object-Relational Mapping u ORM)
 - **Soluciones ORM como JPA, Hibernate, EclipseLink, Entity Framework, NHibernate, etc.**

Patrones en la arquitectura de acceso a datos

- *Table Data Gateway*
- *Row Data Gateway*
- *Active Record*
- *Data Mapper*

Table Data Gateway

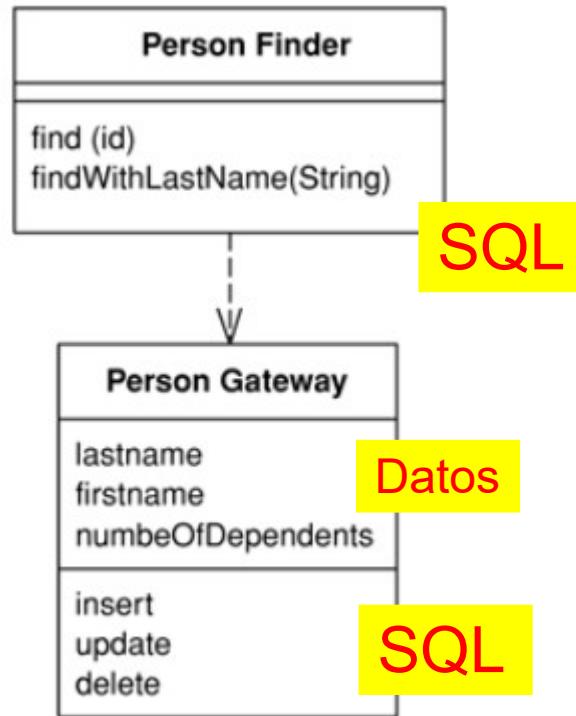
- Un objeto actúa como pasarela a una tabla, **manejando todas las filas de una tabla. Una instancia para todas las filas.**
 - Una **pasarela** por tabla/vista/consulta
- Acoplar SQL y la lógica no es una buena solución.
 - Table Data Gateway **separa** todo el SQL para acceder a una tabla o vista: **SELECT, INSERT, UPDATE y DELETE**
 - Generalmente sin estado
 - Devolviendo: *Data Transfer Objects, Value Object* o en el peor caso *Record Set*
 - Problema: **no hay identidad por lo que casi siempre se debe pasar un id**



Similar a DAO

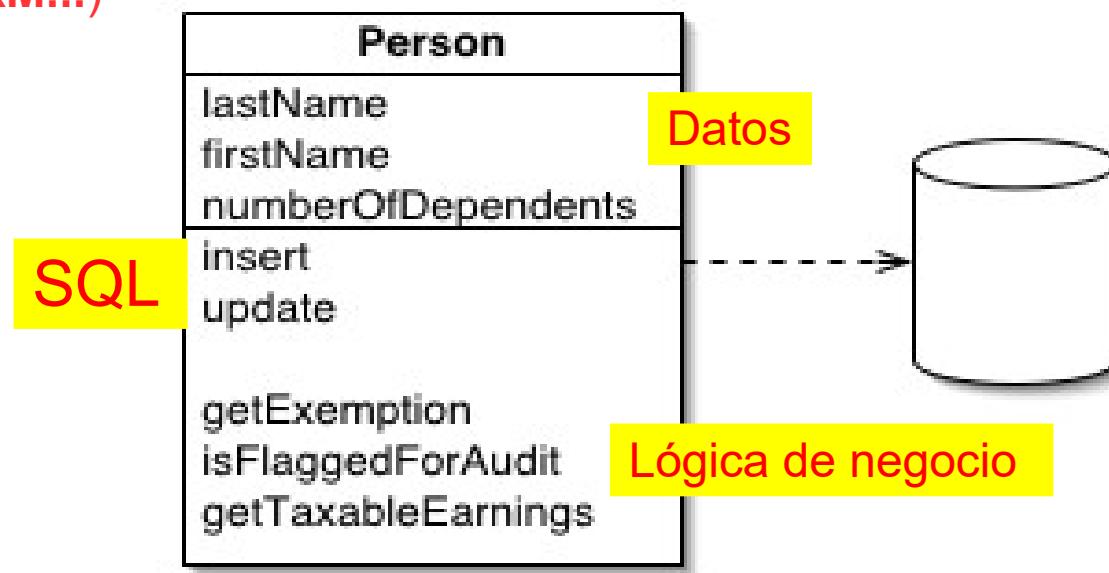
Row Data Gateway

- Un objeto actua como pasarela a un registro. **Una instancia por fila.**
 - Desacopla el acceso a BD.
- Propociona **objetos equivalentes al registro en la BD** pero que pueden ser accedidos con los mecanismos del lenguaje.
- Sólo contiene acceso a datos **no lógica de negocio** (frente a *Active Record*)



Active Record

- Un **objeto por fila**
 - Responsable de guardar y cargar datos, y de **lógica de dominio (negocio)**
 - **Datos + Persistencia (SQL) + Lógica de negocio**
 - Fácil de aplicar si el modelo de objetos y la BD son isomorfos (**¿lo son?...**)
 - Si se quiere utilizar relaciones, colecciones, herencia, etc. se complica... y se tiende a usar el patrón *Data Mapper* (y entonces nos movemos a un **ORM...**)

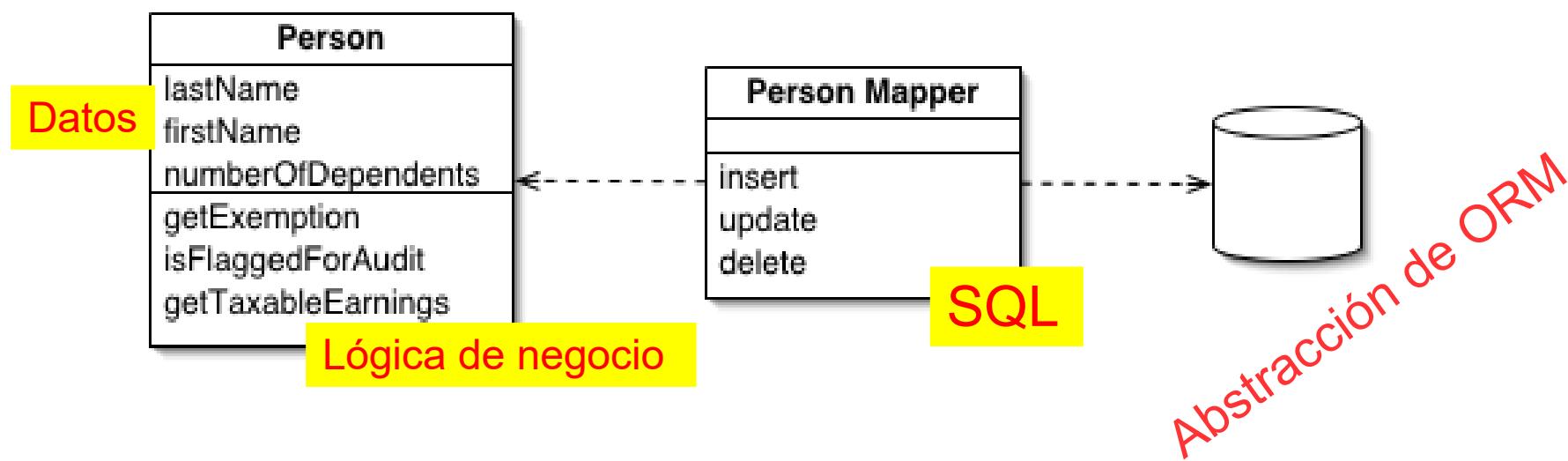


Active Record

- Métodos a incorporar en un *Active Record*:
 - Construir una instancia de un *Active Record* desde una fila resultado de una consulta SQL
 - Construir una nueva instancia a partir de la última inserción en la tabla.
 - Buscadores estáticos para envolver consultas SQL y devolver objetos *Active Record*
 - Actualizar la base de datos e insertar los datos de un *Active Record*
 - *Getters* y *setters* sobre los campos
 - Añadir métodos con lógica de negocio

Data Mapper

- Capa que mueve datos entre objetos y la base de datos, **manteniéndolos independientes**
 - Incluso del *mapper*...
- Complejidad en la transferencia de datos.
 - **Responsable de transferir datos entre las dos capas y aislarlas.**
- A mayor complejidad de la BD, más motivos para usarlo
 - En caso contrario utilizar los anteriores patrones



Otros patrones

- *Unit of Work*
 - Consistencia de los datos: ¿cuándo leer y escribir datos?
 - Mantener registro de datos: limpios y sucios
- *Identity Map*
 - No cargar el mismo datos más que una vez. Tablas de objetos indexados por id. para reutilizar y evitar duplicados.
- *Lazy Load*
 - Optimización en la carga de datos, sólo cargando los datos necesarios en cada momento (LAZY vs. EAGER)
- Otros problemas
 - Relaciones
 - Herencia
 - Transacciones
 - Etc.

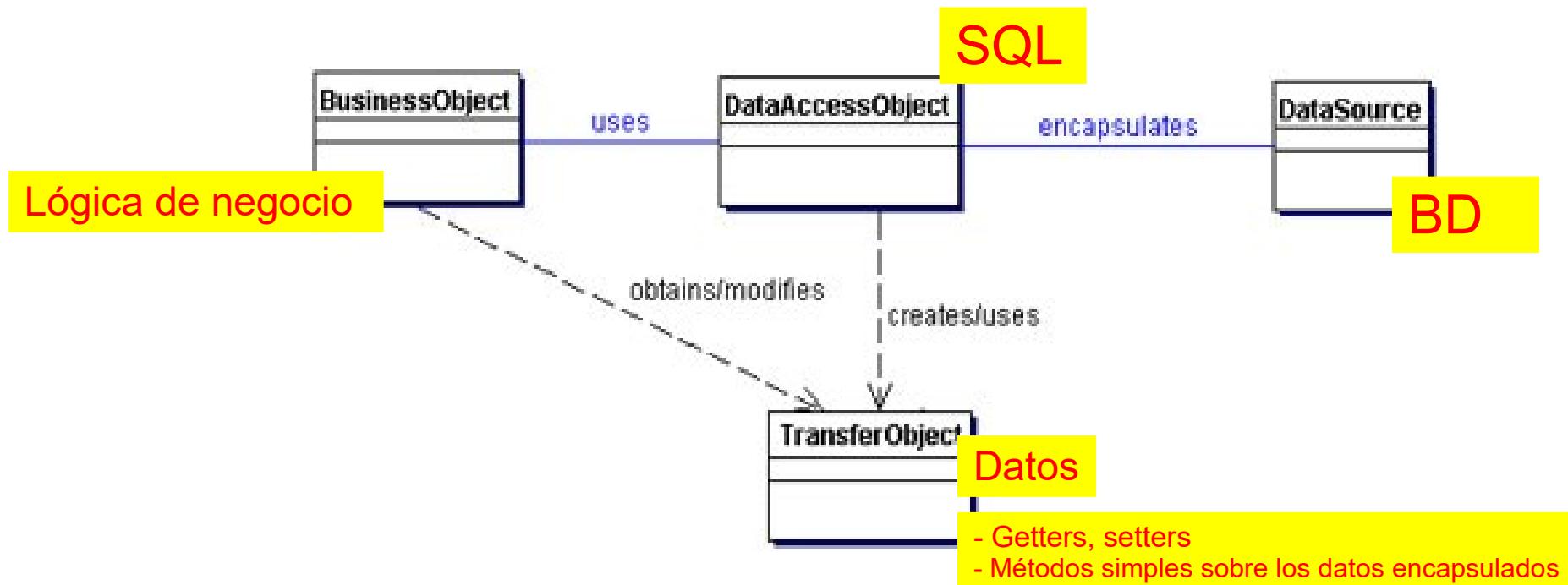
Caso de estudio

Patrón de Diseño Data Access Object (DAO)

- **Contexto:**
 - El acceso a los datos varía según la fuente.
 - En función del **tipo de almacenamiento** (BD Relacional, BD OO, ficheros planos, ficheros XML, JSON, etc.)
 - En función del **proveedor**
- **Solución:**
 - Usar un Data Access Object (DAO) para abstraer y **encapsular** el acceso a la fuente de datos
 - El DAO **gestiona** la conexión con la fuente de datos para realizar las operaciones CRUD

Caso de estudio: PD DAO

- Patrón de diseño
 - En aplicaciones de servidor (J2EE como origen histórico... ahora JEE)
 - Generalizable a otros contextos
 - Participantes:

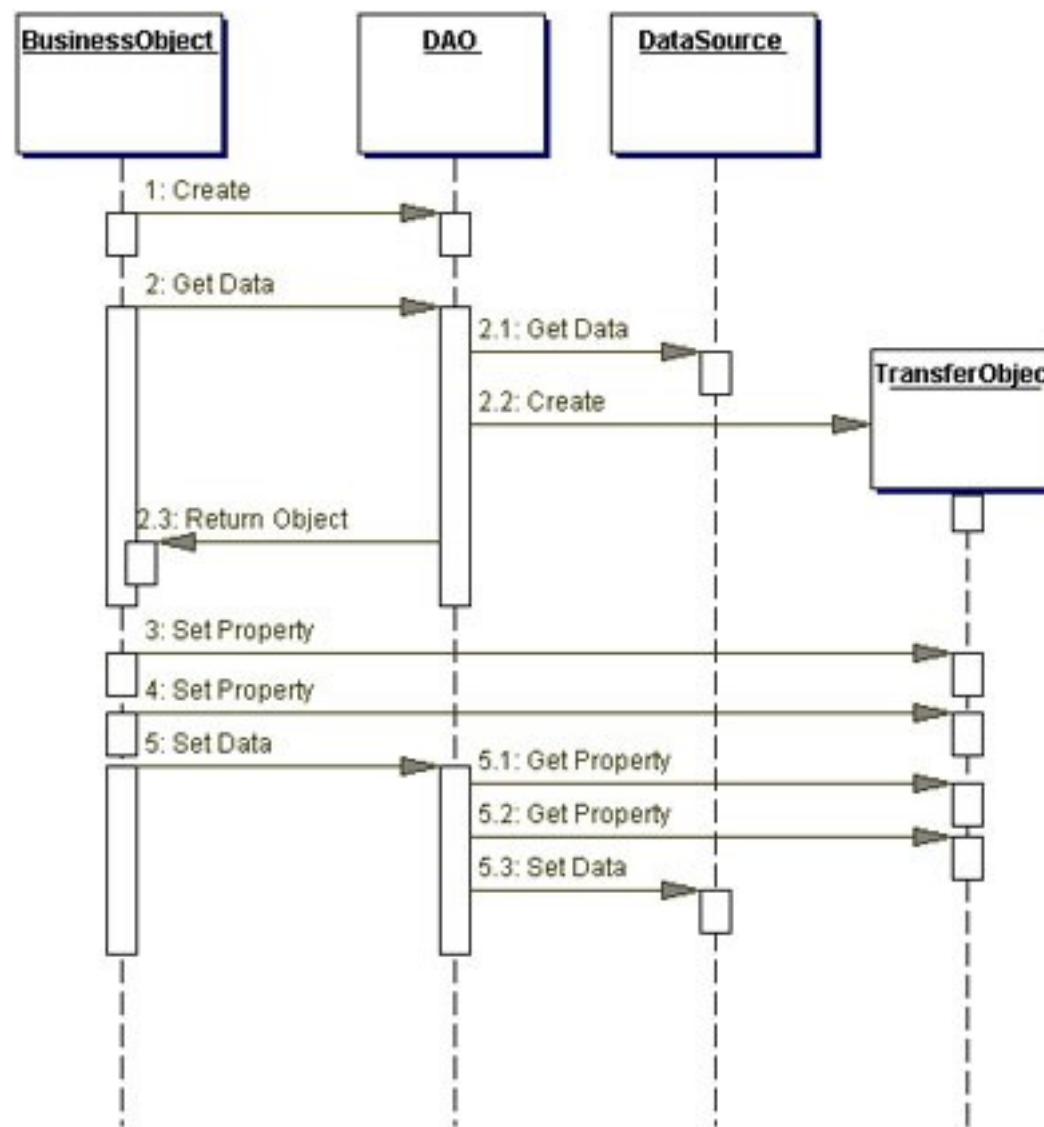


Caso de estudio: PD DAO

- **BusinessObject**
 - Representa al **cliente**. Necesita acceso a la fuente de datos para recuperar y almacenar información.
- **DataAccessObject**
 - Abstacta la capa subyacente de **acceso a datos**, haciendo transparente al cliente dicho acceso. Encapsula el SQL embebido.
- **DataSource**
 - Implementación de la **fuente de datos**.
 - Ej: RDBMS, OODBMS, XML, ficheros planos, etc.
 - Pueden ser otros sistemas (mainframes, sistemas legados, servicios (B2B), servidores LDAP, etc.
- **TransferObject**
 - Para almacenar y transferir datos. Atributos con *getters* y *setters*.

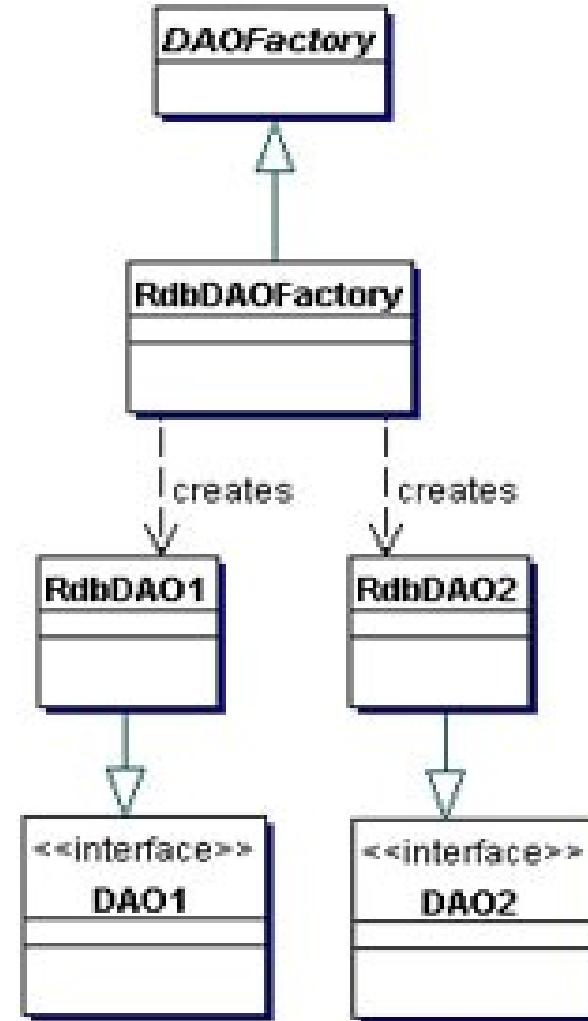
Caso de estudio: PD DAO

- Modelo de interacción



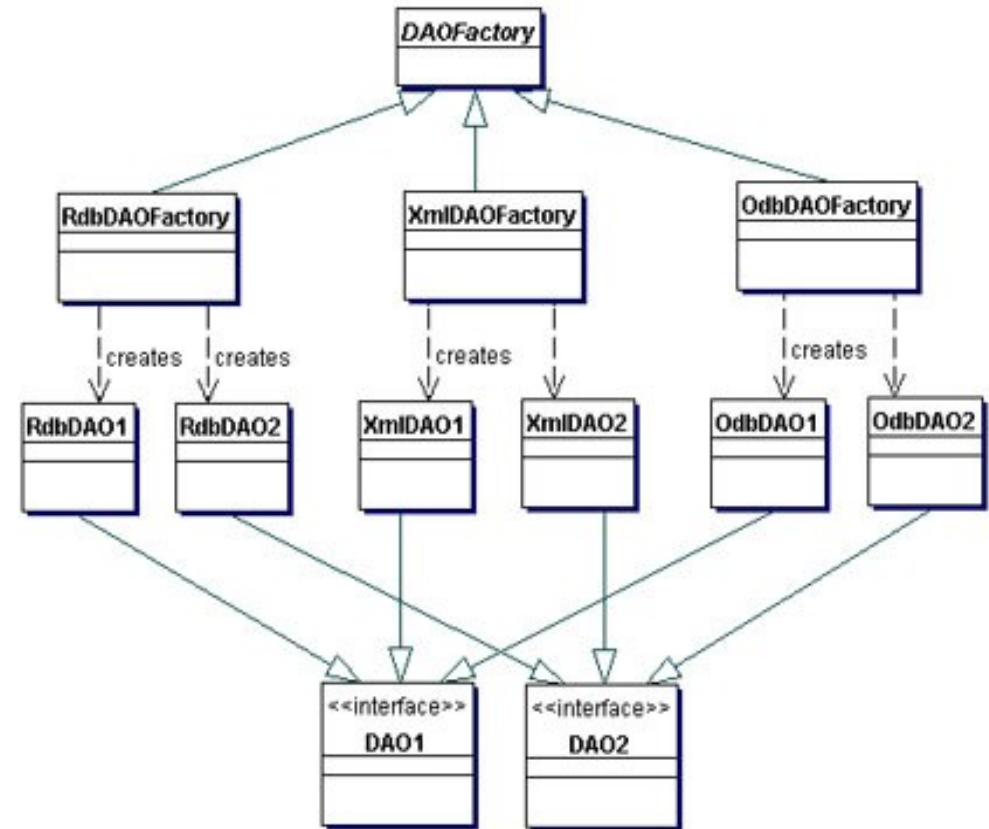
Caso de estudio: PD DAO

- **Estrategias**
 - PD Factory Method
 - Si la fuente de datos no cambia entre diferentes implementaciones
 - Reduce en nº de DAOs



Caso de estudio: PD DAO

- **Estrategias**
 - PD Abstract Factory Method
 - La fuente de datos **SÍ** cambia entre diferentes implementaciones
 - Mayor nº de DAOs



Caso de estudio: PD DAO

- **Ventajas**

- **Más transparencia:** el cliente no están acoplados al tipo de fuente de datos.
- **Facilita la migración:** el cliente no cambia. Cambia la capa de DAO. El uso de patrones y estrategias facilita este cambio.
- **Clientes más simples:** la complejidad de acceso a datos reside en el DAO
- **Centraliza el acceso a datos en una capa:** la capa DAO aisla todo el acceso facilitando el mantenimiento.

- **Desventajas**

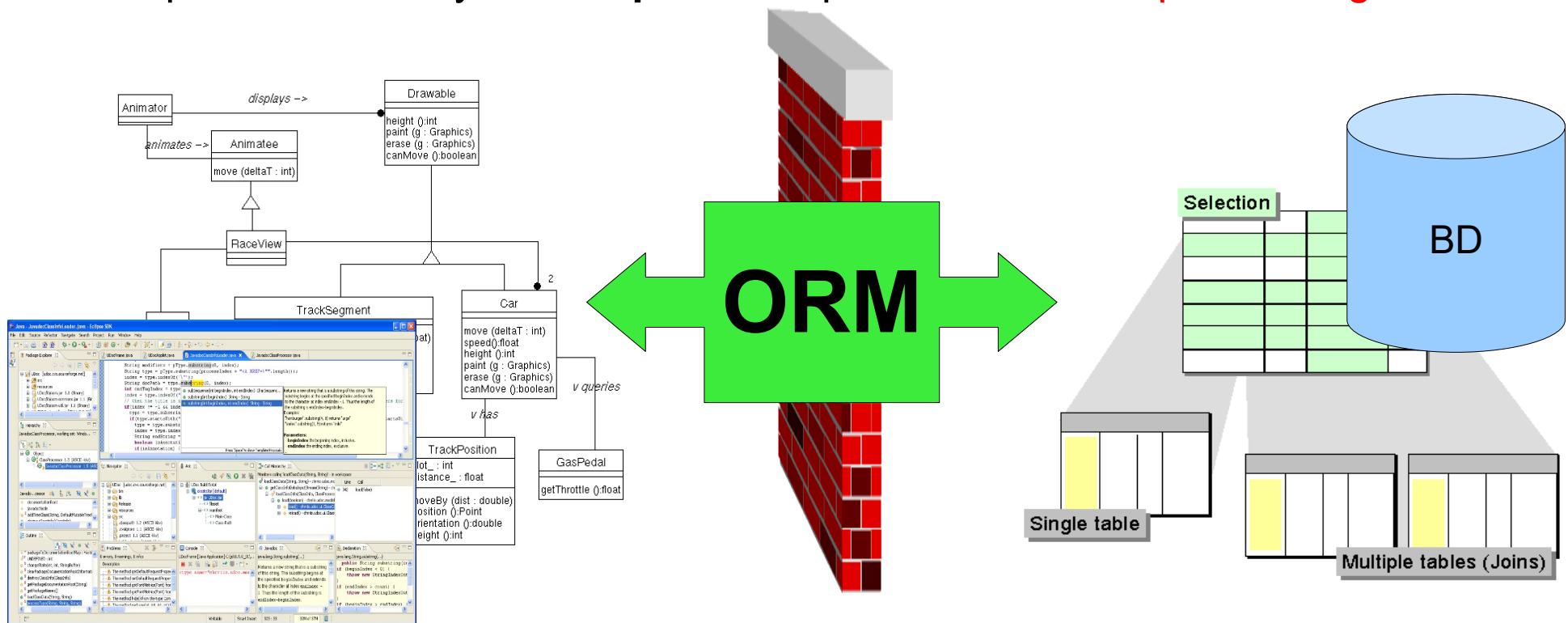
- **Añade una capa extra:** la capa DAO crea una capa adicional entre el cliente y la fuente de datos.
- **Necesita de un diseño de la jerarquía de herencia:** el uso de estrategias con fábricas y fábricas abstractas obliga a un diseño adicional de jerarquías.

Solución: Object-Relational Mapping (ORM)

- Framework de persistencia
 - Definición: "*un conjunto de tipos de propósito general, reutilizable y extensible, que proporciona funcionalidad para dar soporte a objetos persistentes.*"
 - Objetivo: "*Tiene que traducir objetos a registros (o alguna otra forma de datos estructurados como XML) y guardarlos en una BD, y traducir los registros a objetos cuando se recuperan.*"

Solución: Object-Relational Mapping (ORM)

- Gestor entre el **modelo de objetos** y el **proveedor de persistencia** (BD)
 - Puente entre el **modelo del dominio** y el **modelo relacional**
 - Aportando la mayor **transparencia** posible... no siempre conseguida



Solución: ORM

- **Principios**
 - Aplicación escrita en base al modelo del dominio (NO a la BD, aunque pivotando sobre ella...)
 - Con conocimiento de BBDD (no se pueden ignorar, **no lo hace todo por nosotros**)
 - Poco intrusivo pero NO transparente
 - Usar la funcionalidad solo necesaria
 - Entidades locales pero que puedan moverse
 - Con vista a resolver otros problemas:
 - entornos distribuidos con objetos móviles... (Sistemas Distribuidos)

Migración hacia ORM

- Aplicaciones con JDBC
 - ¡No hay equivalencia de ResultSet o RowSet en entidades del ORM!
 - *Mismatch* entre acceso indexado a un array de objetos y acceso por columnas a valores
 - Estrategia: aislar **operaciones JDBC** y reemplazar en bloque aplicando patrones de arquitectura vistos
 - Consultas SQL de JDBC no encajan con objetos del dominio
 - No es tan fácil como reescribir de SQL al lenguaje de consulta del ORM (Ej: JPQL en JPA... siguientes temas)
- Si se ha usado el PD **Data Access Object** es más fácil realizar esta migración
 - Recordatorio: "Aisla las operaciones JDBC en una interfaz"
 - El uso de **Transfer Object** también simplifica la migración de filas de un ResultSet/RowSet a una clase del dominio en ORM (Ej: entidades en JPA... siguiente tema)
 - Fácil si la relación es 1 a 1 con *Transfer Object* a @Entity
 - Con elemento de grano grueso (resultado de varias tablas) se puede conectar con el resultado de una consulta (Ej: expresiones de construcción en JPA)

Migración hacia ORM

- Si se hace uso de fachadas o servicios de sesión que aislan el uso de la persistencia
 - Fácil migración de DAO a ORM

```
public class ProjectServiceBean implements SessionBean {  
    private ProjectHome projectHome; // EJB EntityBean  
    private EmployeeHome empHome; // EJB EntityBean  
    //...  
  
    public void addEmployeeToProject(int projectId, int empId)  
        throws ApplicationException {  
        try {  
            Project project = projectHome.findByPrimaryKey(projectId);  
            Employee emp = empHome.findByPrimaryKey(empId);  
            project.getEmployees().add(emp);  
        } catch (FinderException e) {  
            throw new ApplicationException(e);  
        }  
    }  
    // ...  
}  
  
@Stateless  
public class ProjectServiceBean {  
    //...  
    @PersistenceContext(name="EmployeeService") // Dependency Injection  
    private EntityManager em;  
  
    public void addEmployeeToProject(int projId, int empId)  
        throws ApplicationException {  
        Project project = em.find(Project.class, projId);  
        if (project == null)  
            throw new ApplicationException("Unknown project id: " + projId);  
        Employee emp = em.find(Employee.class, empId);  
        if (emp == null)  
            throw new ApplicationException("Unknown employee id: " + empId);  
        project.getEmployees().add(emp);  
        emp.getProjects().add(project);  
    }  
    // ...  
}
```

DAO (acceso a fuente de datos, realmente usando EJB EntityBean en el ejemplo)

Reemplazado por el gestor de entidades del ORM



Bibliografía

- [Alur et al., 2003] Alur, D., Crupi, J., and Malks,D. (2003) **Core J2EE Patterns: Best Practices and Design Strategies**. 2nd Edition. Prentice Hall PTR. ISBN: 0131422464
- [Yang, 2013] Daoqi Yang (2013) **Java Persistence with JPA 2.1**. Outskirts Press.
- [Fowler, 2003] Fowler, M. (2003) **Patterns of Enterprise Application Architecture**. Addison-Wesley Professional; 1st edition ISBN-10: 0321127420 ISBN-13: 978-0321127426
 - Chapter 3. Mapping to Relational Databases
 - Chapter 10. Data Source Architectural Patterns
- [Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol (2013) **Pro JPA 2. A Definitive Guide to Mastering the Java Persistence API**. Apress. 2nd edition
- [Keith et al., 2018] Mike Keith, Merrick Schincariol and Massimo Nardone (2018) **Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence**. Apress. 3rd edition
- [Larman, 2003] Larman, C. (2003) **UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado**. Prentice Hall. 2^a Edición.
 - Capítulo 34. Diseño de un framework de persistencia con patrones



Documentación adicional

- Core J2EE Patterns – Data Access Object
 - <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

Licencia

Autores: Raúl Marticorena & Mario Martínez & Pablo García
Área de Lenguajes y Sistemas Informáticos
Departamento de Ingeniería Informática
Escuela Politécnica Superior
UNIVERSIDAD DE BURGOS

2022



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirlIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <http://creativecommons.org/licenses/by-nc-sa/4.0/>



Grado en Ingeniería Informática

APLICACIONES DE BASES DE DATOS

TEMA 4

Sección 2. JPA: Introducción y Entidades

Docentes:

Raúl Marticorena

Mario Martínez

Pablo García



Índice de contenidos

1. INTRODUCCIÓN.....	3
2. OBJETIVOS.....	3
3. JPA: INTRODUCCIÓN.....	3
3.1 Concepto.....	3
3.2 Características.....	4
3.3 Arquitectura.....	4
4. ENTIDADES.....	5
4.1 Acceso a atributos y propiedades.....	7
4.2 Tipos Java.....	9
4.3 Carga de valores.....	10
4.4 Clases embebidas.....	10
4.5 Claves primarias: simples vs. compuestas.....	11
4.6 Claves primarias: generación de valores únicos.....	14
4.7 Tipos enumerados.....	15
4.8 Tipos temporales.....	16
4.9 Datos transitorios.....	17
4.10 Anotaciones de columna.....	17
5. RESUMEN.....	18
6. GLOSARIO.....	18
7. BIBLIOGRAFÍA.....	18
8. RECURSOS.....	18



1. Introducción

En la siguiente sección se introduce el concepto de la API de persistencia para Java – Java Persistence API (JPA desde ahora) – y sus propiedades básicas, mostrando la arquitectura general con la que se trabajará en el resto de la asignatura. A continuación se revisan los conceptos básicos para el adecuado ajuste (*mapping*) entre clases/objetos (en LPOO) y tablas/registros (en BD relacionales) a través del concepto de entidades y el uso de anotaciones Java.

2. Objetivos

- Conocer JPA, sus propiedades básicas y arquitectura.
- Conocer las reglas básicas de mapeo entre clases (entidades) y tablas, junto con sus correspondientes objetos y registros respectivamente.

3. JPA: Introducción

A lo largo de los años siempre existió un gran problema con el desajuste entre el mundo de los objetos y el mundo de las bases de datos relacionales. Este problema se ha denominado tradicionalmente en la literatura como *impedance mismatch*. Básicamente, no encajan dichos modelos: la conversión de objetos a filas en las tablas y viceversa no es directa, ni trivial.

Existiendo ciertas soluciones comerciales de herramientas ORM (Object-Relational Mapping) en el mercado, se intentó la definición de una API general para distintos proveedores de frameworks de persistencia (e.g. en Java: Hibernate, EclipseLink, TopLink, etc.)¹.

3.1 Concepto

Como solución al problema planteado surge el concepto de JPA o API de persistencia para Java. Su versión 1.0 nace dentro de la especificación de EJB 3.0, a la sombra del concepto fallido de los *Entity JavaBeans*, para posteriormente tomar cuerpo de especificación independiente en versiones 2.0 y posteriores.

JPA se apoya desde su primera versión en el concepto de Plain Old Java Object (POJO). Como su propio nombre indica, son clases Java muy simples ("planas", con atributos y métodos *getter/setter* inicialmente aunque no exclusivamente) a las que se adornan con algún aditamento sintáctico del lenguaje (anotaciones) para que pasen a tomar un rol diferente en la aplicación (entidades en el caso de JPA).

JPA 2.0 añade alguna funcional adicional como aumento en las capacidades de mapeo, mayor flexibilidad en el modo de acceso al estado de las entidades, mejora de la API de consulta (JPQL) y creación de criterios de consulta dinámicos. En la versión 2.1 se añaden nuevas mejoras, pero de menor calado, como conversores, mejoras en las consultas, etc. La especificación 2.2 incluye algunas mejoras adicionales.

¹ En cierto sentido, una evolución similar a JDBC como API de acceso a bases de datos, con una cierta independencia del proveedor particular de SGBD.



Básicamente se trata de utilizar el concepto de **metadatos** para dirigir el desarrollo: evolucionando desde soluciones con descriptores en ficheros XML externos (solución heredada de la especificación previa de EJB) al uso intensivo de anotaciones (desde que fueran incluidas en Java 1.5).

3.2 Características

Para trabajar con JPA es necesario trabajar con POJOs. Estas clases, que siguen el modelo de JavaBeans (modelo muy antiguo en Java) con atributos y sus correspondientes métodos `get` y `set`, tienen una correspondencia con una entidad persistente y con su correspondiente tabla en la base de datos. Es decir, **sus datos en memoria se deben recuperar y guardar en la base de datos de una manera transparente.**

Para ello no se hace uso de la API JDBC directamente. Se añaden anotaciones a la clase que ayudan al **framework** de persistencia a realizar las correspondientes operaciones SQL.

Inicialmente es desconcertante trabajar con los POJOs, por la ausencia inicial de código JDBC y SQL embebido, aunque como veremos posteriormente, esta aparente simplicidad se complica con el mayor uso de anotaciones y atributos asociados.

Por otro lado señalaremos que inicialmente se plantea también la movilidad de POJOs para su uso en contextos distribuidos y desarrollo de aplicaciones cliente/servidor y/o web (recordad que están dentro de la especificación Java Enterprise Edition o JEE), aunque esta es una cuestión que no se abordará en esta asignatura.

La idea fundamental es que el uso de JPA no sea intrusivo, lo cual no quiere decir que sea transparente, pero exige poca modificación a las clases y en particular al cuerpo de los métodos de las clases, donde reside el código ejecutable.

Para la realización de consultas, JPA añade dos soluciones:

- Una primera basada en su propio lenguaje de consulta, aunque influenciada por SQL: Java Persistence Query Language (JPQL), heredando características de la API EJB QL, ya existente en la especificación EJB.
- Y como segunda opción una navegación en el modelo de objetos, sin utilizar los conceptos de columnas, claves primarias/foráneas y *joins*. Para ello utiliza una API particular denominada Criteria API.

Finalmente, JPA ofrece ciertas convenciones y directrices en el desarrollo, que facilitan su uso:

- Doble solución con la configuración XML vs. Anotaciones, a elección del programador.
- Gran peso a los valores por defecto: siguiendo el lema de "**convención sobre configuración**".
- Facilidad para su integración en servidores de aplicaciones (JEE).
- Posibilidad de funcionamiento independiente de servidores e integración de pruebas automáticas.

3.3 Arquitectura

Con todos estos conceptos introducidos, podemos mostrar a continuación la arquitectura general con la que se trabajará en la asignatura (ver Ilustración 1):



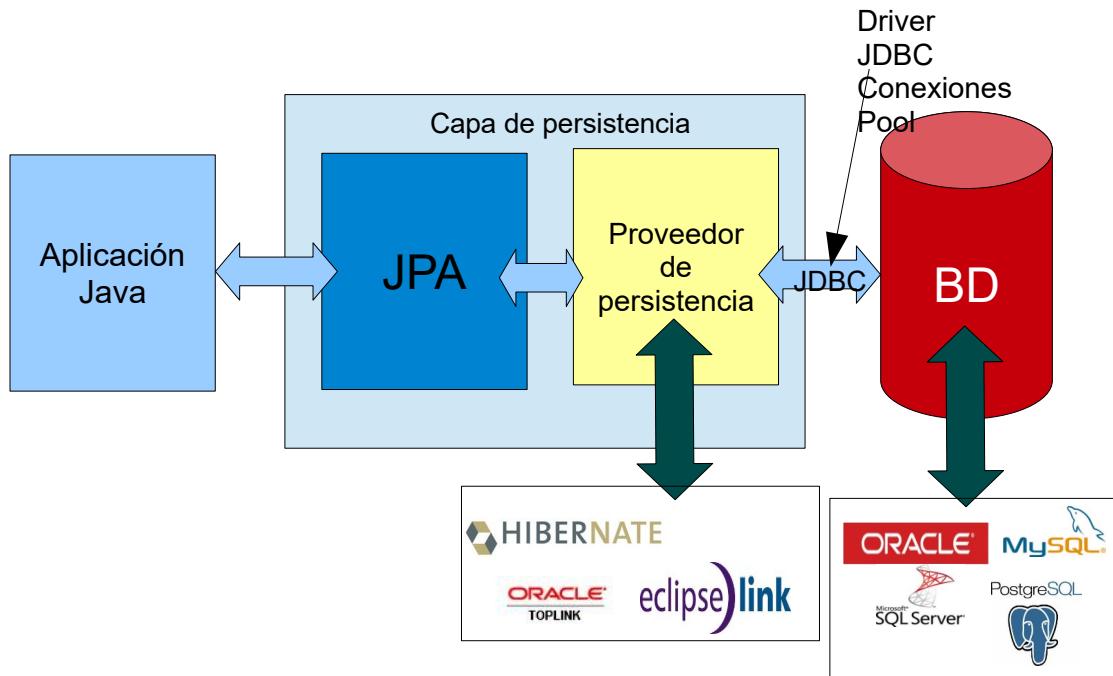


Ilustración 1: Arquitectura general con JPA en la asignatura

Como programadores, **nuestro trabajo** consistirá en construir el código correspondiente al bloque **Aplicación Java**. Este bloque está formado por el código del conjunto de entidades persistentes (mapping con la base de datos), clases de utilidad, clases de patrones de acceso a datos como DAO y el código que propiamente implementa la transacción o lógica de negocio.

En nuestro código utilizaremos solo elementos de **JPA** que permitirán enganchar con el *framework* de persistencia concreto, pero en la medida de lo posible, **sin atarnos a una solución concreta de implementación** (respecto al *framework* de persistencia).

La **Capa de persistencia** es implementada a través de **JPA** como interfaz (abstracta) – siguiendo la metáfora de enchufes en una instalación eléctrica real – y conectando en tiempo de ejecución con una implementación concreta de un **proveedor de persistencia** (e.g., Hibernate en nuestro caso en prácticas) A su vez al proveedor de persistencia se le indica declarativamente (en un fichero externo) el *driver JDBC* y los parámetros de la cadena de conexión a la BD.

Así pues, **si utilizamos correctamente JPA en nuestro código**, y no utilizamos código dependiente del proveedor (e.g. Hibernate) ni de la base de datos concreta (e.g. Oracle), el código es en mayor medida **independiente**, y podríamos posteriormente ejecutar nuestra aplicación con una nueva combinación de proveedor/SGBD (e.g., EclipseLink con MySQL).

Esto no siempre es fácil ni inmediato. Cada *framework* y SGBD incluyen ajustes, soluciones y optimizaciones propias, no siempre incluidos en la especificación JPA. Se produce la típica batalla entre “soluciones muy generales pero no óptimas”, frente a la “optimización con una solución concreta” (atándonos a proveedores particulares, con los riesgos que tiene).

En esta asignatura, y en la parte correspondiente a JPA se intentará tomar, en la medida de lo posible, el primer enfoque, siendo conscientes de los pros y contras.

4. Entidades

Inicialmente abordaremos el problema de la persistencia entendiendo que una instancia u objeto (generado a partir de una clase) tiene una correspondencia 1 a 1 con una fila o registro de una tabla. Los



atributos se corresponden con columnas y viceversa. Las relaciones entre entidades introducen un nivel de complejidad superior que será abordado más adelante en otras secciones del tema.

Para convertir un POJO a entidad, se siguen las siguientes normas:

- La clase será anotada con `@Entity` (`javax.persistence.Entity`).
- Tendrá un constructor sin argumentos o constructor no-arg (`public` o `protected`).
- Sin atributos ni métodos persistentes con modificador `final`.
- Si se pasa como objeto remoto implementa `Serializable` (realmente se pasa por valor).
- Pueden heredar de entidades y no entidades.
- Los atributos persistentes NO pueden ser públicos: siempre se accede con `getters`, `setters` y métodos de negocio.

A continuación se muestra un ejemplo de cómo convertir una clase `Employee` en una entidad persistente, añadiendo un par de anotaciones (ver Código 1):

```
package examples.model;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Employee {

    @Id // Primary Key (PK)
    private int id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    // more setters and getters using name, salary, etc.
    @Override
    public String toString() {
        return "Employee id: " + getId() + " name: " + getName() + " salary: " + getSalary();
    }

    // other methods...
}
```

Código 1: Ejemplo básico de entidad JPA

Como se puede comprobar en el ejemplo, la clase no es “*muy diferente*” a una clase tradicional en Java: atributos, métodos `get`, `set` y algún método típico como `toString()`. Si queremos que los objetos de `Employee` se correspondan con filas en la tabla `EMPLOYEE`, hay que modificar la clase con anotaciones (como podemos ver no es del todo transparente, pero el trabajo necesario tampoco es grande).



Simplemente **añadiendo la anotación @Entity justo delante de la declaración de la clase**, ya estamos indicando que la clase se transforma en una entidad persistente. Para finalizar el trabajo debemos indicar que el atributo `id` se corresponde con la clave primaria `ID` en la tabla `EMPLOYEE`. Sin más.

En este ejemplo se sigue el lema de "convención sobre configuración": se asume que el nombre de la clase/entidad y de la tabla coinciden, y que el nombre del atributo `id` y de la clave primaria en la tabla también coinciden. Dependiendo del grado de similitud de coincidencia entre nombres, nos tocará realizar más o menos trabajo.

Si no existiese esta coincidencia, hay que ampliar la información en la anotación, como se muestra en el siguiente ejemplo (ver Código 2). En dicho ejemplo se asume que la tabla y que los atributos tienen nombres diferentes o acortados (e.g. la tabla tendría la siguiente definición `EMP(EMP_ID (PK), NAME, SAL, COMM)`).

```

@Entity
@Table(name="EMP")      // Example 1: table name EMP
public class Employee { ... }

@Entity
@Table(name="EMP", schema="HR") // Example 2: table and schema in Oracle
public class Employee { ... }

@Entity
@Table(name="EMP", catalog="HR") // Example 3: table and catalog in SQLServer
public class Employee { ... }

// Example 4 changing column names
@Entity
@Table(name="EMP")
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    @Column(name="SAL")
    private long salary;
    @Column(name="COMM")
    private String comments;
    ...
}

```

Código 2: Ejemplo básico de entidad JPA sin correspondencia de nombres

JPA no dispone de anotaciones especiales para vistas, utilizando la misma anotación `@Table` (hasta la fecha no se ha incluido ninguna anotación especial para vistas). En estos casos seguimos limitados a las operaciones posibles sobre una vista, típicamente (aunque no exclusivamente) solo consultas, ignorándose las operaciones que modifiquen datos. Como curiosidad, el framework Hibernate añade una anotación `@Immutable`, que no es parte del estándar. En esta asignatura nos centraremos en el trabajo con tablas.

4.1 Acceso a atributos y propiedades

Llegado este punto puede surgirnos la duda: ¿cómo accede el *framework* a los valores de los atributos para crear la correspondiente SQL y mover los datos entre ambos "mundos"?

El acceso puede ser de dos formas (y no exclusivo):

1. Accediendo a los atributos:

- A través de reflexión, en tiempo de ejecución.
- Anotando el atributo que debe tener modificador distinto a `public`.



- Ej: `@Id private int id;`

2. Accediendo a propiedades:

- Con los correspondiente métodos `getX` y `setX` donde `x` se sustituye por el nombre del atributo (y métodos `isX` para consulta con booleanos).
 - Los métodos deben ser `public` o `protected`.
 - La anotación se coloca sobre el método `get`.
- Ej: `@Id public int getId() { return id; }`

Sin embargo en JPA se permite un acceso mixto, utilizando ambos mecanismos a la vez (atributo vs. propiedad/método). También se permite que las subclases redefinan el método de acceso (heredado) (e.g. `@Access(AccessType.FIELD)`).

Estos mecanismos no se aplican a datos temporales no persistentes (`transient` en Java) o propiedades marcadas con `@Transient`. En el siguiente ejemplo (ver Código 3), se muestra el uso combinado de estas soluciones.

```

@Entity
@Access(AccessType.FIELD)
public class Employee {
    public static final String LOCAL_AREA_CODE = "613";
    @Id private int id;
    @Transient private String phoneNum;
    ...
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getPhoneNumber() { return phoneNum; }
    public void setPhoneNumber(String num) { this.phoneNum = num; }

    @Access(AccessType.PROPERTY) @Column(name="PHONE")
    // exception of general case using field
    protected String getPhoneNumberForDb() {
        if (phoneNum.length() == 10)
            return phoneNum;
        else
            return LOCAL_AREA_CODE + phoneNum;
    }

    protected void setPhoneNumberForDb(String num) {
        if (num.startsWith(LOCAL_AREA_CODE))
            phoneNum = num.substring(3);
        else
            phoneNum = num;
    }
    ...
}

```

Código 3: Acceso básico combinado con atributos vs. propiedades

Inicialmente se ha indicado un acceso por atributo anotando la clase (`@Access(AccessType.FIELD)`). Dicho acceso se puede observar por ejemplo en el atributo `id`.

Sin embargo este comportamiento se cambia en el acceso al número de teléfono por parte de la base de datos (atributo `phoneNum`). En este caso, se accede al atributo a través de dos métodos: `getPhoneNumberForDb` (método anotado con `@Access`) y `setPhoneNumberForDb`. Además, al no existir coincidencia de nombre con los métodos, se ha indicado explícitamente la columna en la tabla – `@Column(name="PHONE")`. Esto permite “interceptar” los datos entre el modelo de objetos y la BD, realizando las operaciones de transformación sobre el número de teléfono. En este ejemplo concreto, cuando se quiere guardar el número de teléfono del objeto, en la BD, a través del método `get`, se añade o no el prefijo local si procede. Cuando se lee el valor de la BD a través del método `set`, se elimina el prefijo del valor a almacenar en el objeto, cuando procede. De esta forma ambos métodos **interceptan** y



transforman los valores. Cuando sean necesarias este tipo de transformaciones, es necesario utilizar el acceso con propiedades.

4.2 Tipos Java

Las restricciones en cuanto al uso de tipos Java en JPA se establece en:

- Tipos primitivos y clases de envoltura (*wrappers*) como `Long`, `Character`, etc.
- Cadenas de texto como `String`.
- Números largos como `java.math.BigInteger` y `java.math.BigDecimal`.
- Fechas como `java.util.Date` y `java.util.Calendar`
- Tipos de datos SQL para fecha y hora, como `java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`.
- Tipos serializables.
- Arrays de `byte[]`, `Byte[]`, `char[]` y `Character[]` (para tipos de datos `BLOB` o `CLOB`).
- Tipos enumerados Java.
- Clases embebidas propias de JPA (se verán posteriormente).

Para colecciones de datos se recomienda el uso de interfaces Java, y **preferiblemente con genericidad**:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

Nota: la especificación JPA 2.2 incluye ya algún método adicional que permite también la integración de la interfaz `java.util.Stream` incluida en la versión 8 de Java.

Las colecciones de tipos básicos de Java o embebidos, usan la anotación `@ElementCollection` con atributos de anotación `targetClass` (con tipo embebido se indica, pero se omite con tipos Java ya definidos) y `fetch` (por defecto `LAZY`, frente a `EAGER`). El atributo `fetch` se explicará en el siguiente apartado.

En el siguiente ejemplo (ver Código 4), se muestran distintos usos de los tipos de datos colección, remarcando el uso **no recomendado** de las soluciones sin genericidad (tipos *raw*) como el uso de `Collection` o `ArrayList`, sin dar un parámetro genérico actual:

```
@Entity
public class Person {
...
@ElementCollection(fetch=EAGER)
protected Set<String> nickname = new HashSet<String>();
...

@ElementCollection(targetClass=Phone, fetch=EAGER)
protected Collection phones = new ArrayList(); // Not recommendable

@ElementCollection(fetch=LAZY)
protected List<Address> addresses = new ArrayList<Address>();
}
```

Código 4: Ejemplo de tipos de datos colección con JPA



Estas colecciones de elementos, se utilizan en relaciones uno a varios con valores cuya vida está ligada a la entidad fuerte de la que dependen, y cuyos datos están almacenados en otra tabla. En posteriores secciones del tema se profundizará sobre las relaciones más habituales entre entidades fuertes, más habituales (i.e., uno a uno, uno a varios o varios a varios).

4.3 Carga de valores

En JPA se permite indicar al *framework* si queremos que los valores se carguen de manera inmediata (ansiosa o `EAGER`) o por el contrario de manera diferida (perezosa o `LAZY`) al cargar los datos de una entidad.

La implementación de `LAZY` por parte del proveedor de persistencia es **opcional** (puede que lo haga o no) pero en el caso de `EAGER` es **obligatoria**.

Tiene sentido su uso si no es necesario traer todo el dato (o datos) inicialmente, y se puede esperar a su acceso real por parte de la aplicación para recuperar el valor de la base de datos (pudiera ser que nunca se accede al dato y por lo tanto supone un ahorro de recursos). El propio *framework* controla el acceso al atributo y la posible necesidad de acceder al valor.

Esto es particularmente útil con datos binarios o de texto grandes – donde se combina con la anotación `@Lob` – (tipos `BLOB`, `CLOB`, etc. en base de datos), ver Código 5, o con las filas relacionadas a otra entidad persistente.

```
@Entity  
public class Employee {  
    @Id  
    private int id;  
    @Basic(fetch=FetchType.LAZY)  
    @Lob @Column(name="PIC")  
    private byte[] picture;  
    // ...  
}
```

Código 5: Ejemplo de vinculación perezosa de un dato binario grande (`BLOB`)

Nota: la anotación `@Basic` se puede utilizar **opcionalmente** sobre cualquier propiedad persistente o variable de instancia de tipo primitivo, envoltorio (wrappers), `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, enumerados, y cualquier otro tipo que implemente `Serializable` (ver especificación de JPA 2.1). Permite redefinir el valor por defecto de `fetch` (con valor por defecto `Fetch.EAGER`) y también especificar el valor booleano de la pista (`hint`) `optional` para indicar si el valor podría valer `null` o no (ignorado su uso para tipos primitivos). Dado su carácter opcional, solo se utilizará en aquellos casos en los que sea necesario realmente redefinir el valor por defecto, normalmente del `fetch`.

4.4 Clases embebidas

Se entiende por una clase embebida a aquellas **entidades especiales que no tienen identidad, ligadas a la entidad fuerte que las contiene**, permitiendo su uso recursivamente (una clase embebida puede estar compuesta de otras). Por otro lado pueden establecer relaciones con otras entidades y colecciones.

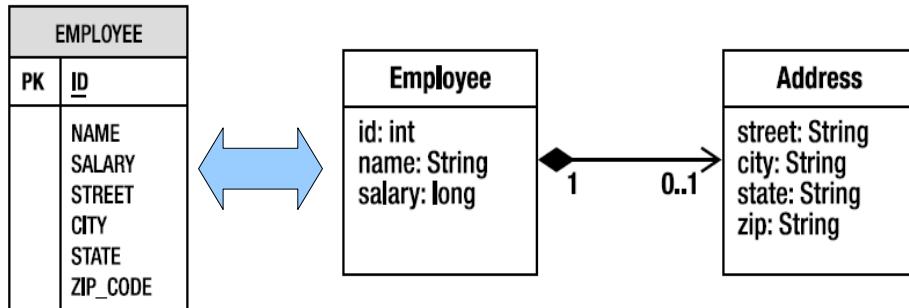
La anotación utilizada en la **declaración** de estas clases es `@Embeddable` en lugar de `@Entity`.

Para su **uso** (no requerido) en otras clases se utiliza `@Embedded`.

Tomemos como ejemplo el Dibujo 1, donde una tabla `EMPLOYEE` contiene campos que definen su dirección. Esto se transforma en el diagrama de clases en una composición entre `Employee` y `Address`



(rombo negro) con navegabilidad de Employee hacia Address (punta de flecha), pero no al revés. La multiplicidad 0..1 indica que no se registra dirección para todo empleado, pudiendo estar sus valores vacíos o nulos.



Dibujo 1: Ejemplo de clase embebida. Figura extraída de [Keith & Schincariol, 2013]

La dirección no tiene como tal un identificador o clave primaria, su identificación (y su vida) está ligada al empleado correspondiente, y por lo tanto su rol en JPA es el de una clase (tipo) embebido.

Para representar esta situación en JPA se utilizaría el siguiente código parcial (ver Código 6):

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded private Address address;
    // ...
}

@Embeddable @Access (AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
    // ...
}
  
```

Código 6: Uso de clases (tipos) embebidos

4.5 Claves primarias: simples vs. compuestas

Un concepto fundamental en las bases de datos, es el de **clave primaria** en nuestras entidades. La correspondencia de dicho concepto en JPA, como ya se ha visto, es a través de la anotación `@Id`.

Los tipos que se pueden utilizar para la clave primaria son tipos primitivos, *wrappers*, `java.lang.String`, `java.util.Date`, `java.sql.Date`, `java.math.BigDecimal` y `java.math.BigInteger`.

Aunque se pueden utilizar, se desaconseja el uso `float`, `double`, `Float`, `Double` y `BigDecimal`, dado que debido a errores de redondeo, pueden generar problemas a la hora de realizar comparaciones con el método `equals`².

Sin embargo, la clave primaria no siempre se compone de un único campo/atributo (clave simple), dándose la posibilidad de que sea compuesta por varios.

Para construir las **claves compuestas** se dan dos variantes o posibilidades:

1. Clase identidad: `@IdClass`
2. Identificador embebido: `@EmbeddedId`

En ambas soluciones, las clases que componen la “identidad” y que se usan como clave primaria tienen unas reglas a cumplir específicamente:

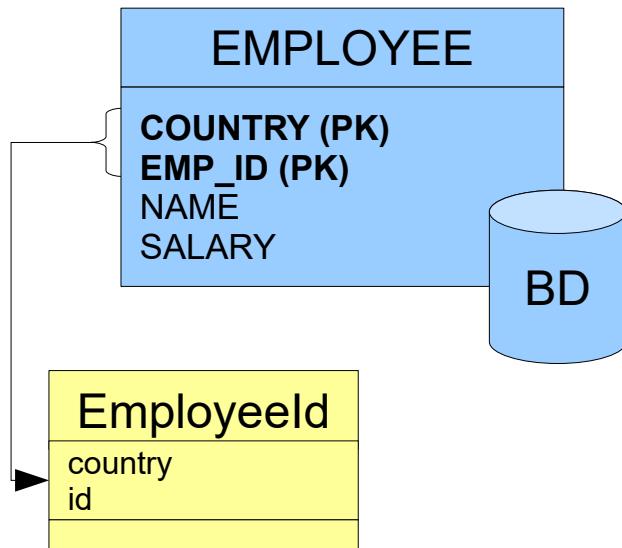
- Son clases públicas, con propiedades públicas o protegidas.

2 Recordad que el método `equals` se hereda de `Object`, se suele redefinir y permite comparar el estado de los objetos, no sus identidades. Es el método utilizado por JPA para comparar objetos entre sí.



- Tienen un constructor sin argumentos público.
- Redefinen `hashCode` y `equals`.
- Implementan la interfaz `Serializable`.
- Vinculadas a:
 - Atributos de la entidad (coinciden en nombre) en la `@IdClass`.
 - Clase embebida (`@EmbeddedId`).

A continuación, partiendo de la situación mostrada en Dibujo 2, donde se muestra la tabla `EMPLOYEE` con una clave compuesta formada por `COUNTRY` y `EMP_ID`, se resuelve su implementación con ambas soluciones a continuación.



Dibujo 2: Tabla con clave primaria compuesta y clase identidad asociada

En primer lugar, utilizando **clases identidad**. En el Código 7, se muestra la clase/entidad `Employee`. Se indica que tiene una clave compuesta implementada en una clase identidad, a través de la anotación de clase `@IdClass`, con valor adicional la clase que implementa dicha clave compuesta (`EmployeeId.class`).

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {

    @Id private String country;
    @Id @Column(name="EMP_ID")
    private int id;

    private String name;
    private long salary;
    // ...
}
```

Código 7: Clase con clave primaria compuesta implementada en clase identidad

La clase que implementa la clave compuesta (`EmployeeId`) (ver Código 8), debe cumplir las reglas generales ya expuestas, y además **debe existir** una correspondencia uno a uno entre sus atributos y los atributos marcados en la entidad persistente con `@Id`.



En nuestro ejemplo concreto los atributos, `country` e `id` en `Employee` tienen sus correspondientes réplicas en la clase `EmployeeId`, uno a uno. Como curiosidad señalar la ausencia de anotaciones JPA en la clase identidad.

```
public class EmployeeId implements Serializable {
    private String country;
    private int id;
    public EmployeeId() {} // no-args constructor, mandatory
    public EmployeeId(String country, int id) {
        this.country = country;
        this.id = id;
    }
    // without setters... IMMUTABLE
    public String getCountry() { return country; }
    public int getId() { return id; }

    // utility methods for identity... (inherited and overridden from Object)
    @Override
    public boolean equals(Object o) {
        return ((o instanceof EmployeeId) &&
                country.equals(((EmployeeId)o).getCountry()) &&
                id == ((EmployeeId)o).getId());
    }
    @Override
    public int hashCode() {
        return country.hashCode() + id;
    }
}
```

Código 8: Clase identidad para clave primaria compuesta

Existe una segunda solución, usando **identificadores embebidos**. En esta solución (ver Código 9) se utiliza un atributo (**y sólo uno**) en la clase `Employee` de tipo embebido, indicándose su papel de clave primaria con la anotación `@EmbeddedId`.

A continuación se implementa la clase embebida siguiendo las reglas generales para las claves compuestas. En este caso la **clase embebida** sí contiene anotaciones JPA.

```
@Entity
public class Employee {

    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;
    // ...
}

@Embeddable
public class EmployeeId implements
Serializable {
    private String country;
```

$$\text{@Column(name="EMP_ID")}$$

$$\text{private int id;}$$

$$\text{public EmployeeId() {}}$$

$$\text{public EmployeeId(String country, int id) {}}$$

$$\text{this.country = country;}$$

$$\text{this.id = id;}$$

$$\text{// Getter methods, equals() and hashCode()}$$

$$\text{// implementations, are the same as @IdClass example}$$

Código 9: Clase con clave primaria compuesta implementada con clase embebida



4.6 Claves primarias: generación de valores únicos

Un problema habitual con las claves primarias es la generación de valores únicos. En un SGBD se suele incorporar algún mecanismo que facilita su generación, y por lo tanto, es necesario estudiar cómo se integran las distintas soluciones dentro de JPA.

La solución adoptada es indicar en la clave primaria, la **estrategia** de generación utilizada por el SGBD. Se proporcionan cuatro:

1. AUTO: el proveedor de persistencia proporciona el id. Sin importar la estrategia. Suele necesitar permisos DBA (para crear recursos). **Nota: se usa en producción y prototipos pero no en explotación. No utilizar.**
2. TABLE: utiliza una tabla con dos columnas, 1) nombre del "generador de secuencia" y 2) valor (último id). Cada generador es una fila de la tabla, simulando el uso de secuencias. Se puede especificar el nombre de la tabla, del generador, columna clave y valor. Además se pueden dar parámetros adicionales como el valor inicial (`InitialValue`, valor por defecto 0) y una caché de valores (`AllocationSize`, valor por defecto 50). Es una solución portable, pero no recomendable si nuestra BD soporta secuencias o autonuméricos.
3. SEQUENCE: mecanismo interno de la BD (e.g., Oracle, PostgreSQL) para generar identificadores. Tiene un alto rendimiento.
4. IDENTITY: mecanismo interno de la BD generalmente a través de valores autonuméricos (e.g., MySQL o SQLServer). Menor rendimiento que las secuencias.

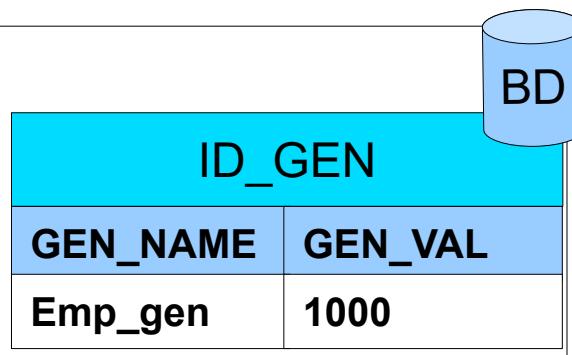
A continuación se presentan los correspondientes códigos de ejemplo para los cuatro casos. En el primer caso (ver Código 10) se delega en el proveedor de persistencia el mecanismo de generación de valores. Aunque es muy simple se recomienda no utilizarlo.

```
@Entity  
public class Employee {  
    @Id @GeneratedValue(strategy=GenerationType.AUTO)  
    private int id;  
    // ...  
}
```

Código 10: Generación con AUTO

En el segundo caso (ver Código 11), se utiliza una tabla auxiliar para generar los identificadores. Es necesario indicar el nombre de la tabla (e.g. `ID_GEN`) y nombre del generador (e.g. `Emp_Gen`) así como los nombres de las columnas que contienen el nombre del generador y el valor actual (e.g. `GEN_NAME` y `GEN_VAL` respectivamente). Es una solución muy portable, puesto que no depende de mecanismos particulares del SGBD.

```
// default values taken by provider...  
@Id @GeneratedValue(strategy=GenerationType.TABLE)  
private int id;  
  
// all values...  
@TableGenerator(name="Emp_Gen", table="ID_GEN",  
pkColumnName="GEN_NAME",  
valueColumnName="GEN_VAL",  
initialValue=1000,  
allocationSize=100)  
@Id @GeneratedValue(generator="Emp_Gen")  
private int id;
```



Código 11: Generación con TABLE



```
// default values taken by provider...
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
private int id;

// settings with sequence name
@SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
@Id @GeneratedValue(generator="Emp_Gen")
private int getId;
```

Código 12: Generación con SEQUENCE

La tercera opción es utilizar el concepto de secuencias que existe en algún SGBD. Genera automáticamente nuevos identificadores, resolviendo conflictos propios del uso concurrente. Ejemplo típico es Oracle, utilizado en esta asignatura.

Finalmente, la última opción, usando una estrategia de identidad, generalmente resuelta por el SGBD a partir de la generación de valores auto-numéricos, como por ejemplo en MySQL o SQLServer.

```
// default values without generator
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

Código 13: Generación con IDENTITY

En general, el uso de un SGBD concreto suele influir mucho (quizás demasiado) en la utilización de una u otra estrategia. Dado que en la asignatura los ejemplos y ejercicios se resuelven con Oracle, se utilizará preferiblemente la solución basada en secuencias, por simplicidad y rendimiento. Aunque no sea la solución más portable.

4.7 Tipos enumerados

Tomamos una definición clásica de tipo enumerado como "*tipo de datos con un conjunto de valores acotado y que implícitamente tienen también asignado un valor ordinal*". Este valor ordinal puede ser utilizado el almacenamiento y acceso al valor.

Ejemplos típicos son los días de la semana (e.g., Lunes, martes, miércoles, etc.) o los palos de la baraja (e.g., Oros, Copas, Espadas y Bastos).

En Java, desde su versión 1.5 se incluyó este concepto de tipo enumerado y por lo tanto se utilizará para almacenar valores que se ajusten a la definición previa en JPA (ver Código 14).

Habitualmente se utiliza un valor almacenado de tipo entero en la BD, para almacenar el valor correspondiente del tipo enumerado (solución basada en posición). Esta solución es óptima en espacio, pero origina una serie de problemas si el tipo enumerado cambia o se reordena.

```
public enum EmployeeType {
    FULL_TIME_EMPLOYEE,      // ordinal 0
    PART_TIME_EMPLOYEE,       // ordinal 1
    CONTRACT_EMPLOYEE         // ordinal 2
}

@Entity
public class Employee {
    @Id private int id;
    private EmployeeType type; // valor por defecto @Enumerated(EnumType.ORDINAL)
    // ...
}
```

Código 14: Ejemplo de tipo enumerado con ordinal y uso en atributo de entidad



Como alternativa al uso del ordinal, se plantea el uso del texto correspondiente en la BD. Esta solución no sufre de los problemas por cambio y reordenación, pero es menos eficiente y sigue siendo vulnerable a cambio en los textos descriptivos. Para llevar a cabo esta solución se utilizan las anotaciones @Enumerated y valor enumerado EnumType (ver Código 15).

```
// Using previous definition of EmployeeType

@Entity
public class Employee {
@Id
private int id;
@Enumerated(EnumType.STRING)
private EmployeeType type;
// ...
}
```

Código 15: Ejemplo de tipo enumerado con texto y uso en atributo de entidad

Si se no se utiliza la anotación @Enumerated o no se especifica el valor EnumType, se supone el valor por defecto, que es EnumType.ORDINAL con valor entero.

Se recuerda además que aunque los enum en Java pueden incluir atributos y métodos adicionales (son básicamente clases Java), **JPA no da soporte a estos elementos adicionales**.

4.8 Tipos temporales

A la hora de trabajar con valores de tipo fecha y hora, se establecen mapeos especiales entre atributos y campos. El mapeo adicional se produce en los campos genéricos de tipo java.util.Date y java.util.Calendar, **no siendo** necesario en los tipos de datos específicos java.sql.Date, java.sql.Time ni java.sql.Timestamp.

Para los dos tipos de datos genéricos, en frameworks como Hibernate, se toma valor por defecto timestamp con nanosegundos. Para cambiar dicho valor se deben añadir la anotación @Temporal. Junto con la anotación, se debe indicar el valor correspondiente de mapeo SQL según se quiera:

- Almacenar solo la fecha (día, mes y año) con TemporalType.DATE.
- Almacenar solo el tiempo (sin nanosegundos) con TemporalType.TIME.
- Almacenar fecha y hora (con nanosegundos) con TemporalType.TIMESTAMP.

En el Código 16 se muestran algunos ejemplos de uso con TemporalType.DATE (su uso con los otros valores es similar sintácticamente, se omite por brevedad).

```
@Entity
public class Employee {
@Id
private int id;
@Temporal(TemporalType.DATE)
private java.util.Calendar dob;
@Temporal(TemporalType.DATE)
@Column(name="S_DATE")
private java.util.Date startDate;
// ...
}
```

Código 16: Ejemplo de mapeo especial para tipo de dato temporal

En la especificación 2.2 de JPA se incluye el soporte de los tipos del paquete java.time incluidos en la versión Java 8, como parte de su "Date and Time API". Algun framework como Hibernate da soluciones particulares desde su versión 5, pero no son objeto de estudio ni uso en la asignatura, dado que no son parte del estándar ni portables.



4.9 Datos transitorios

Nos podemos encontrar con atributos que existen en el modelo de objetos **pero que no tienen un reflejo directo (ni deben) en la base de datos**. Estos datos solo están en memoria y se pierden finalmente al eliminar el objeto.

En Java se indica esto, utilizando un modificador `transient`, que es una palabra reservada del lenguaje (ojo, tampoco se serializan estos atributos).

En JPA, se proporciona un mecanismo equivalente adicional con la anotación `@Transient`, evitando que se realicen mapeos sobre estos atributos en la base de datos. Puede ser útil con datos calculados o derivados, que no tienen un reflejo directo en la base de datos.

4.10 Anotaciones de columna

Como se ha mostrado en algún ejemplo previo, existe una anotación adicional para el mapeo de atributos denominada `@Column`.

Esta anotación da **información adicional** relativo al mapeo físico con la base de datos y es particularmente útil si se quiere generar el DDL de las tablas (crearlas) a partir de la información recogida en el modelo de clases, pero no cumplen un papel de validadores. Dichas comprobaciones, o bien se realizan a nivel de la base de datos (mediante las *constraints*) o bien mediante el uso adicional de APIs de validación. **Esto es meramente informativo para la generación del DDL y útil para ingeniería inversa desde el modelo de clases hacia la base de datos** (pero en esta asignatura siempre trabajaremos desde la base de datos hacia el modelo de clases).

Por ejemplo se permite la inclusión de información adicional sobre valores nulos permitidos (`nullable`), unicidad (`unique`), longitud para cadenas (`length`) o decimales en numéricos (`scale`).

Sin embargo, su uso sí es obligado cuando los **nombres de los atributos no coinciden con los nombres de los campos correspondientes en la tabla**, de manera similar al uso de `@Table`. Por ejemplo:

```
@Entity
@Table("EMP") // suponemos que la tabla se llama distinto que la entidad
public class Employee {
    @Id
    @Column(name = "IDEN") // suponemos que el campo se llama distinto que el atributo
    private int id;
    @Column(name = "NAME", nullable = false, length = 150) // información adicional sobre el campo
    private String nombre;
    ...
}
```

Código 17: Ejemplo de mapeo adicional con `@Column`

Existen dos atributos adicionales para indicar la posible inserción (`insertable`) o actualización (`updatable`) de la columna (valores booleanos `true/false`) por parte del proveedor. Esto permite configurar, en cierta manera, valores de solo lectura. Pero esto no impide que los valores no puedan ser modificados temporalmente en memoria. Así pues, se aplica esto solo en el momento de traducir los cambios al correspondiente SQL enviado a la base de datos, sin lanzar ninguna excepción de aviso (aunque esto puede ser dependiente del proveedor).

Dado que en esta asignatura, se partirá generalmente de una base de datos ya definida, con el DDL correspondiente ya creado, solo se insistirá en esta anotación `@Column` cuando sea estrictamente necesario. En este mismo sentido, no se profundizará en anotaciones de mapeo similares, como `@UniqueConstraint`, `@Index` o `@ForeignKey` quedando fuera del objetivo de esta asignatura.



5. Resumen

En esta sección se han planteado el concepto, propiedades y arquitectura básica de JPA. Por otro lado se han planteado las reglas básicas de mapeo de entidades JPA, entre el modelo de objetos y la base de datos, con el mapeo básico de atributos, teniendo en cuenta los tipos Java, las políticas de carga de valores (ansiosa vs. perezosa) y el uso de clases embebidas.

Se ha abordado el uso de claves primarias simples y compuestas, la generación de identificadores únicos para dichas claves, y algún uso más avanzado como el de tipos enumerados, tipos temporales, datos transitorios y anotaciones de columna.

Sin embargo, en esta sección se ha simplificado el problema, afrontando el problema del mapeo entidad-tabla, uno a uno. En la práctica sabemos que las entidades están relacionadas entre sí y no son algo aislado. En la siguiente sección se describen las soluciones de mapeo para la relaciones.

6. Glosario

API: Application Programming Interface o interfaz de programación de aplicaciones.

JPA: Java Persistence API

POJO: Plain Old Java Object u objeto Java plano.

SGBD: Sistema Gestor de Bases de Datos

7. Bibliografía

[Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol. ***Pro JPA 2. A definitive guide to mastering the Java Persistence API (2013)*** Apress. 2nd edition.

[Keith et al., 2018] Mike Keith, Merrick Schincariol, Massimo Nardone. ***Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence APIs.*** (2018) Apress. 3rd edition.

[Oracle, 2017] The Java EE 8 Tutorial (2017). Part VIII. Persistence. Disponible en
<https://javaee.github.io/tutorial/toc.html>

[Oracle, 2014] The Java EE 7 Tutorial (2014). Part VIII. Persistence. Disponible en
<http://docs.oracle.com/javaee/7/tutorial/>

8. Recursos

Especificaciones públicas:

[Oracle, 2013] JSR 338: JavaTM Persistence API, Version 2.1 (2013). Version: Final Release
http://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf

[Oracle, 2017] JSR 338: JavaTM Persistence API, Version 2.2 (2017) Version: Maintenance Release
http://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf

Bibliografía complementaria:



Java Persistence. (2021, September 30). *Wikibooks, The Free Textbook Project*. Retrieved 08:08, April 9, 2022 from https://en.wikibooks.org/w/index.php?title=Java_Persistence&oldid=3992576.



Licencia

Autores: Raúl Marticorena & Mario Martínez & Pablo García

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2022



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <http://creativecommons.org/licenses/by-nc-sa/4.0/>





Grado en Ingeniería Informática

APLICACIONES DE BASES DE DATOS

TEMA 4

Sección 3. JPA: Relaciones

Docentes:

Raúl Marticorena

Mario Martínez

Pablo García

Índice de contenidos

1. INTRODUCCIÓN.....	3
2. OBJETIVOS.....	3
3. RELACIONES.....	3
3.1 One to One (uno a uno) - Unidireccional.....	3
3.2 One to One (uno a uno) - Bidireccional.....	4
3.3 Many to One (Muchos a uno) - Unidireccional.....	5
3.4 One to Many (uno a muchos) – Unidireccional con Join Table.....	7
3.5 One to Many (uno a muchos) – Unidireccional sin Join Table.....	8
3.6 One to Many (uno a muchos) - Bidireccional.....	8
3.7 Many to many (muchos a muchos) – Unidireccional con Join Table.....	11
3.8 Many to many (muchos a muchos) – Bidireccional con Join Table.....	12
3.9 Orden en las asociaciones a varios.....	14
4. HERENCIA.....	14
4.1 Single table.....	15
4.2 Table per class.....	16
4.3 Joined.....	17
4.4 Mixed.....	19
5. OPERACIONES EN RELACIONES.....	21
5.1 Carga de datos en relaciones.....	21
5.2 Mantenimiento de relaciones.....	22
6. CLAVES DERIVADAS.....	23
6.1 Solución con @IdClass.....	23
6.2 Solución con @EmbeddedId.....	24
6.3 Clave primaria compartida.....	25
6.4 Reglas generales.....	26
7. CLASES EMBEBIDAS (REVISIÓN).....	26
8. USO DE COLECCIONES DE ELEMENTOS (REVISIÓN).....	27
9. ANEXO: INTERFACES JAVA EN JAVA.UTIL CON JPA.....	29
9.1 List.....	29
9.2 Map.....	30
10. RESUMEN.....	34
11. GLOSARIO.....	34
12. BIBLIOGRAFÍA.....	34
13. RECURSOS.....	35



1. Introducción

En esta sección se revisa la correspondencia entre las distintas relaciones definidas en el modelo Entidad-Relación (E-R) y relacional (tablas de la base de datos), y su ajuste con las entidades de JPA.

Como se planteó en la sección previa, la correspondencia no es simplemente 1 a 1 entre entidad y tabla (objeto vs. registro), puesto que las relaciones existentes dificultan en gran medida las operaciones en ambos sentidos.

Sin embargo, una vez establecidas las relaciones en las entidades, el *framework* de persistencia hace transparente toda la problemática asociada a la correcta actualización de claves primarias y foráneas, entre los registros de la base de datos.

2. Objetivos

- Conocer las reglas de mapeo entre las distintas anotaciones JPA y las relaciones.
- Aprender a resolver mapeos más complejos como el existente entre relaciones IS_A y las jerarquías de herencia.
- Revisar los problemas adicionales de implementación, como la carga de datos y el mantenimiento de relaciones con colecciones.
- Ampliar el uso de claves compuestas derivadas.
- Revisar el uso de embebidos y colecciones de elementos.

3. Relaciones

Entre las entidades del modelo E-R, o entre las relaciones del modelo relacional, existen elementos que permiten relacionar y navegar entre las tuplas o filas de la base de datos (i.e., claves). Cuando ese concepto se lleva a al mundo de los objetos, nos encontramos con las asociaciones y reglas de navegabilidad en los modelos de clases de UML.

Aunque con muchas similitudes, también existen diferencias, y por lo tanto es necesario conocer las distintas soluciones que se dan en JPA para definir el adecuado ajuste entre ambos mundos.

Para ello, y desde el punto de vista de las clases, se debe tener en cuenta el rol en la asociación, la dirección (navegabilidad) y las multiplicidades (min/max). Desde el punto de vista de las relaciones en la base de datos hablamos de roles, dirección y cardinalidad/ordinalidad respectivamente.

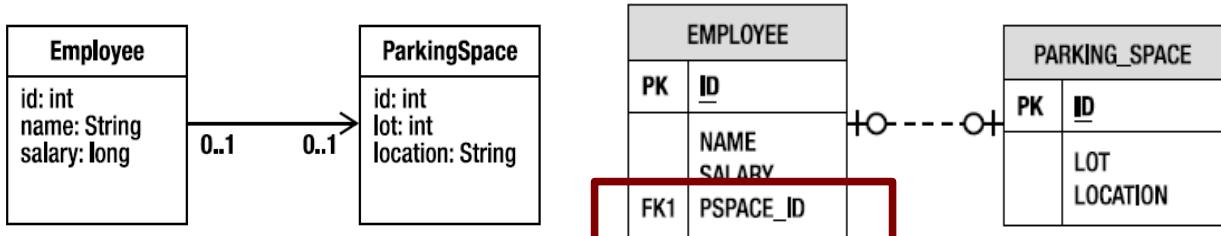
A continuación se van a detallar las reglas básicas de transformación, abordando cada caso con ejemplos.

3.1 One to One (uno a uno) - Unidireccional

Un objeto está relacionado como máximo con un objeto del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado como máximo con un objeto del primer conjunto. **Sólo se puede navegar desde el objeto del primer conjunto al del segundo.**



En el Dibujo 1, podemos ver la correspondencia entre el diagrama de clases y el modelo relacional correspondiente.



Dibujo 1: Ejemplo de One to one - unidireccional. Figura extraída de [Keith & Schincariol, 2009]

Para implementar la navegabilidad en el modelo de clases, un empleado tendrá una referencia a un objeto plaza de aparcamiento, pero una plaza de aparcamiento no tiene ninguna referencia al empleado que la posee. En la base de datos se debe haber establecido una restricción de unicidad sobre la clave foránea.

En JPA se resuelve anotando el atributo o propiedad a través de la que se navega (en este ejemplo la plaza de aparcamiento en la clase empleado):

- Anotando a **nivel lógico** con `@OneToOne`.
- Anotando a **nivel físico** con `@JoinColumn` e indicando el nombre de la columna que cumple el papel de clave foránea en la base de datos.

En el Código 1, se puede ver resuelto el ejemplo:

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```

Código 1: Ejemplo One to One - Unidireccional

Por otro lado, dada la navegabilidad en un solo sentido, se recuerda que la clase `ParkingSpace` no incluye atributo de tipo `Employee` ni ninguna anotación adicional.

3.2 One to One (uno a uno) - Bidireccional

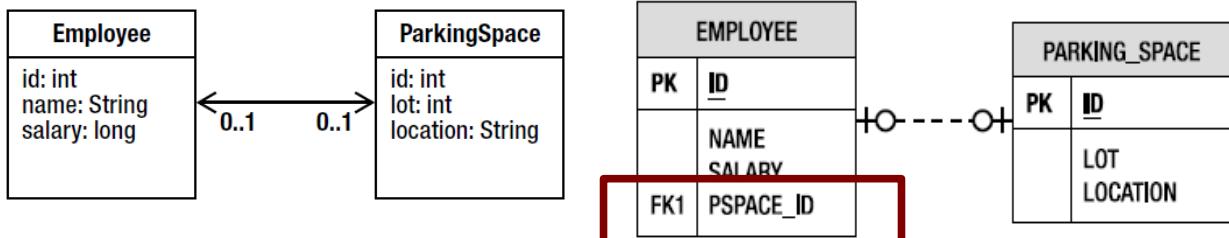
Un objeto está relacionado como máximo con un objeto del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado como máximo con un objeto del primer conjunto. **Se permite la navegación en ambos sentidos desde el objeto del primer conjunto al del segundo, y viceversa.**

En el Dibujo 2, podemos ver la correspondencia entre el diagrama de clases y el modelo relacional correspondiente. Debemos señalar que la única diferencia con el Dibujo 1, es la doble navegabilidad entre ambas clases en el diagrama de clases, siendo el modelo relacional exactamente igual.

Nota importante: en los diagramas extraídos de la bibliografía, la **doble navegabilidad se indica con puntas de flechas en ambos extremos de la asociación**. En otros ejemplos se indicará con la



ausencia de puntas de flechas la doble navegabilidad. Solo en el caso de que haya **una única punta de flecha en un extremo de la asociación** se interpretará como **asociación unidireccional**.



Dibujo 2: Ejemplo de One to one - bidireccional. Figura extraída de [Keith & Schincariol, 2009]

Para implementar la navegabilidad en el modelo de clases, un empleado tendrá una referencia a un objeto plaza de aparcamiento, y una plaza de aparcamiento tendrá una referencia al empleado que la posee. En la base de datos se debe mantener la restricción de unicidad sobre la clave foránea.

Se puede resolver tomando cualquiera de los dos lados como propietario (dependiendo de dónde coloquemos la clave foránea en la base de datos), pero esa decisión afecta posteriormente a dónde se coloca la anotación `@JoinColumn` (**proprietario**) y elemento `mappedBy` (**no-proprietario**) en JPA. El criterio a utilizar es tomar el sentido más frecuente en la consulta y navegación.

En esta solución ambas clases son modificadas. Por lo tanto, de manera similar a los casos previos se resuelve:

- Anotando a **nivel lógico** con `@OneToOne`, tanto en la clase denominada **proprietaria** como en la **no-proprietaria**.
- Anotando a **nivel físico** con `@JoinColumn` e indicando el nombre de la columna que cumple el papel de clave foránea en la base de datos, en la clase **proprietaria**.
- Añadiendo a **nivel lógico** el elemento `mappedBy` a la anotación `@OneToOne`, en la clase **no propietaria**, indicando el **nombre del atributo** en la clase **proprietaria**, a través del cuál se realiza el mapeo.

En el Código 2, se puede ver resuelto el ejemplo:

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}

@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    // ...
}

```

Código 2: Ejemplo One to One - Bidireccional

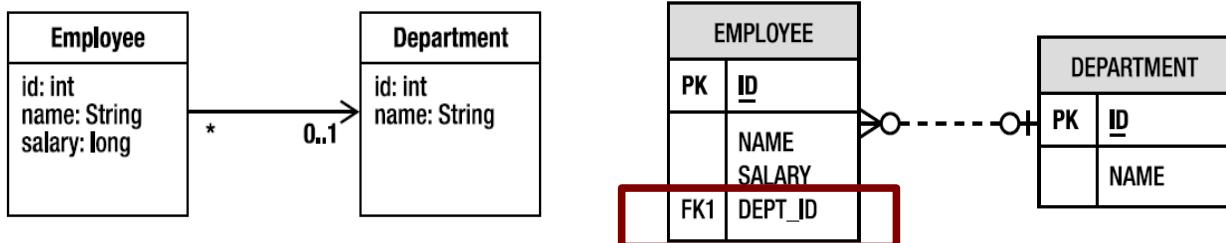
En este caso ambas clases son modificadas, y existe un doble enlace a través de sus atributos.

3.3 Many to One (Muchos a uno) - Unidireccional

Un objeto (hijo) está relacionado como máximo con un objeto del segundo conjunto (padre). Respectivamente, cada uno de estos objetos (padre) del conjunto está relacionado con varios (hijos), pero **sólo se puede navegar desde el objeto hijo al padre**.



En el Dibujo 3, podemos ver la correspondencia entre el diagrama de clases y el modelo relacional correspondiente.



Dibujo 3: Ejemplo de Many to One. Figura extraída de [Keith & Schincariol, 2009]

En este ejemplo, un empleado pertenece (o no) a un departamento como máximo. En un departamento puede haber varios empleados.

La navegabilidad en el diagrama de clases indica que un empleado puede saber a qué departamento pertenece pero un departamento no puede saber qué empleados tiene asignados.

Debemos recordar sin embargo, que **el modelo relacional es bidireccional implícitamente en cuanto a la navegación**. Dada una fila de cualquiera de las dos tablas, se puede conocer la(s) fila(s) relacionada(s) utilizando SQL, SELECT y JOINs. Este **desajuste** (recordad el problema de *impedance mismatch*) se reproduce en el resto de casos.

Para implementar la navegabilidad en el modelo de clases, un empleado tendrá una referencia a un objeto departamento, pero un departamento no tiene ninguna referencia de tipo colección de empleados.

En JPA se resuelve anotando el atributo o propiedad a través de la que se navega (en este ejemplo el departamento en la clase empleado):

- Anotando a **nivel lógico** con `@ManyToOne`.
- Anotando a **nivel físico** con `@JoinColumn` e indicando el nombre de la columna que cumple el papel de clave foránea en la base de datos.

En el Código 3, se puede ver resuelto el ejemplo:

```

@Entity
public class Employee {
    @Id private int id;
    @ManyToOne // logical mapping
    @JoinColumn(name="DEPT_ID") // physical mapping foreign key
    // in table Employee
    private Department department;
    // ...
}

```

Código 3: Ejemplo Many to one - Unidireccional

Por otro lado, dada la navegabilidad en un solo sentido, se recuerda que la clase **Department** no incluye ninguna anotación adicional, dado que no existe atributo a anotar.

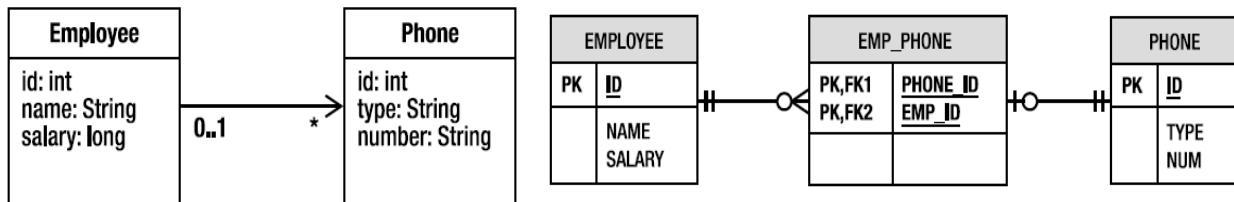
El estudio de la bidireccionalidad se realizará desde el caso de uno a muchos posteriormente.



3.4 One to Many (uno a muchos) – Unidireccional con Join Table

Un objeto está relacionado con un conjunto de objetos del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado como máximo con un objeto del primer conjunto. **Solo se puede navegar desde el objeto del primer conjunto al del segundo.**

En el Dibujo 4, podemos ver la correspondencia entre el diagrama de clases y el modelo relacional correspondiente. En esta solución particular se utiliza en el modelo relacional una **tabla adicional (JOIN TABLE)** que permite realizar el correspondiente enganche entre las filas.



Dibujo 4: Ejemplo de One to many - unidireccional con Join Table. Figura extraída de [Keith & Schincariol, 2009]

En JPA se resuelve, con una diferencia en la clase no propietaria:

- Anotando a **nivel lógico** con `@OneToMany` en la **no-proprietaria**.
- Añadiendo a **nivel físico** con `@JoinTable`, en la clase **no propietaria**, indicando el **nombre de la JOIN TABLE, y las columnas físicas a través de las cuales se realiza el JOIN**.

En este caso, la entidad objetivo (no propietaria) – Employee en el ejemplo – **no incluye una anotación lógica mappedBy, sino que se incluye una anotación física @JoinTable**. Por otro lado, la clase Phone no tiene modificación alguna ni inclusión de anotaciones JPA, puesto que no se puede navegar desde un teléfono al empleado asociado y carece de atributo Employee.

En el Código 4, se puede ver resuelto el ejemplo:

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToMany
    @JoinTable(name="EMP_PHONE",
               joinColumns=@JoinColumn(name="EMP_ID"),
               inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Phone> phones;
    // ...
}
```

Código 4: Ejemplo One to many – Unidireccional con Join Table

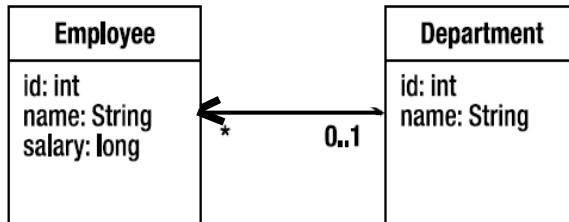
En la anotación `@JoinTable`, es **muy importante** indicar como clave directa (`joinColumns`) las claves de la tabla relativas a la entidad/tabla actual, y como claves inversas (`inverseJoinColumns`) las relativas a las claves de la tabla relacionada. En caso contrario, el funcionamiento será erróneo.



3.5 One to Many (uno a muchos) – Unidireccional sin Join Table

Un objeto está relacionado con un conjunto de objetos del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado como máximo con un objeto del primer conjunto. **Sólo se puede navegar desde el objeto del primer conjunto al del segundo.**

En el Código 5, podemos ver la correspondencia con el diagrama de clases. **En esta solución particular NO se utiliza en el modelo relacional una tabla adicional (JOIN TABLE).**



Dibujo 5: Ejemplo de One to many - unidireccional sin Join Table. Figura extraída de [Keith & Schincariol, 2009]

Se requiere que **la clave foránea se refiera en la tabla objetivo (no-propietaria)**. Es una solución extraña a la hora de implementar.

En el Código 5, se puede ver resuelto el ejemplo:

```
@Entity  
public class Department {  
    @Id private int id;  
    @OneToMany  
    @JoinColumn(name="DEPT_ID")  
    private Collection<Employee> employees;  
    // ...  
}
```

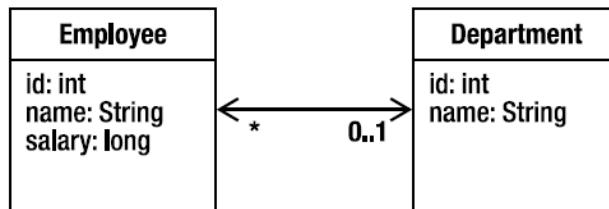
Código 5: Ejemplo One to Many – Unidireccional sin Join Table

Recordemos que la columna DEPT_ID está definida como *foreign key* en la tabla EMPLOYEE.

3.6 One to Many (uno a muchos) - Bidireccional

Un objeto está relacionado con un conjunto de objetos del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado como máximo con un objeto del primer conjunto. **Se puede navegar en ambos sentidos.**

En el Dibujo 6, podemos ver el diagrama de clases correspondiente.



Dibujo 6: Ejemplo de One to many - bidireccional. Figura extraída de [Keith & Schincariol, 2009]



En la práctica se suele definir casi siempre bidireccional, con la tabla con la multiplicidad 0..1 como **no-propietaria**. Para implementar la multiplicidad * (varios) en la clase no-propietaria, se utiliza alguna **interfaz colección** (Tip: no olvidarse de instanciar la clase colección concreta que implementa la interfaz, en el constructor por ejemplo, antes de ser utilizada, o tendremos referencias nulas).

En JPA se resuelve:

- Anotando a **nivel lógico** con `@OneToMany` en la **no-propietaria**.
- Añadiendo a **nivel lógico** el elemento `mappedBy` a la anotación `@OneToMany`, en la clase **no-propietaria**, indicando el **nombre del atributo** en la clase **proprietaria**, a través del cuál se realiza el mapeo.
- Anotando a **nivel lógico** con `@ManyToOne`, en la clase denominada **proprietaria**
- Anotando a **nivel físico** con `@JoinColumn` e indicando el nombre de la columna que cumple el papel de clave foránea en la base de datos, en la clase **proprietaria**.

En el Código 6, se puede ver resuelto el ejemplo:

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
    // method setEmployees(Collection<Employee> emp) {}
    // method Collection<Employee> getEmployees() {}
}

// No se muestra @OneToMany en Employee, se resuelve de la forma ya vista.
```

Código 6: Ejemplo One to Many - Bidireccional

Por otro lado, la clase `Employee` se implementa de la forma ya vista para *many to one* (con anotación `@ManyToOne`) en la Sección 3.3 (ver Código 3) y que incluye un atributo de nombre `department` y tipo `Department`.

Nota muy importante: en las relaciones bidireccionales (y más en particular con las relaciones varios a varios) es importante realizar las actualizaciones **a AMBOS lados por parte del código de nuestra aplicación** (crear correctamente el doble enlace entre referencias). Si se añade a un lado de la relación, debe también añadirse al otro lado (**JPA NO lo hace automáticamente**).

Esto se suele resolver a través de métodos como `add`, `remove` y `set` que ajustan AMBOS lados de la relación. Normalmente con dos estrategias:

- Añadir en ambos lados el código explícito, y evitando posibles llamadas infinitas entre los métodos (e.g., mediante comprobación de pertenencia). Este problema se define como la **"corrupción de objetos"**. Se dispone de un ejemplo de esta estrategia con código en: http://www.java2s.com/Tutorials/Java/JPA/4710__JPA_Query_Join_ManyToOne.htm.
- Añadir el código solo en un lado e invocar al correspondiente `set` adicional solo desde dicho lado (solución generada por ejemplo por el plugin *Eclipse Dali* incluido en nuestro IDE). Esto es mucho más simple, pero condicionado a que el objeto que se añade es de nueva creación y no estaba relacionado con otros. Hay muchas comprobaciones que se omiten (e.g., posibles valores nulos, objetos previamente ya enlazados, etc.)

Se muestra el código de ejemplo (sin incluir anotaciones JPA) a continuación en Código 7, con una asociación bidireccional `@OneToMany` de `Employee` a `Phone` (y viceversa `@ManyToOne` de `Phone` a `Employee`).



```

public class Employee {
    private List<Phone> phones;
    ...
    public void addPhone(Phone phone) {
        if (phone != null && !getPhones.contains(phone)) { // si ese teléfono no estaba
            getPhones.add(phone); // añade el teléfono al empleado...
            if (phone.getOwner() != null) { // si el teléfono ya tenía empleado asociado
                phone.getOwner().getPhones().remove(phone); // se elimina dicha asignación
            }
            phone.setOwner(this); // enlaza el empleado al teléfono
        }
    }
    ...
}

public class Phone {
    private Employee owner;
    ...
    public void setOwner(Employee employee) {
        if (getOwner() != null) { // tenía propietario
            getOwner().getPhones().remove(this); // se elimina dicha asignación
        }
        this.owner = employee;
        if (employee != null && !employee.getPhones().contains(this)) {
            // si el nuevo empleado existe y no tenía ese teléfono
            employee.getPhones().add(this); // añade el teléfono al empleado...
        }
    }
    ...
}

```

Código 7: Ejemplo de código para establecer el “doble enlace” en asociaciones bidireccionales a varios

Si utilizamos el generador de código de Eclipse (*Eclipse Dali*) que generar el modelo de entidades a partir de tablas, podemos observar como genera métodos `add` y `remove` de manera similar (en este caso con la variante simplificada pero menos robusta). En el ejemplo del Código 8, suponemos que estamos en la clase `Grupo`, mostrando los dos métodos que gestionan los equipos de un grupo. En la clase `Equipo`, se completa con el método `setGrupo`.



```
@Entity
public class Grupo implements Serializable {
    ...
    @OneToMany(mappedBy="grupo")
    private Set<Equipo> equipos;
    ...
    // Métodos de una clase Grupo que contiene equipos (varios Equipo)
    public Equipo addEquipo(Equipo equipo) {
        getEquipos().add(equipo);
        equipo.setGrupo(this);
        return equipo;
    }

    public Equipo removeEquipo(Equipo equipo) {
        getEquipos().remove(equipo);
        equipo.setGrupo(null);
        return equipo;
    }
}

@Entity
public class Equipo implements Serializable {
    ...
    @ManyToOne
    @JoinColumns(name="LETRA")
    private Grupo grupo;

    public void setGrupo(Grupo grupo) {
        this.grupo = grupo;
    }
}
```

Código 8: Ejemplo de código generado por Eclipse Dali para garantizar el "doble enlace" en asociaciones a varios

En esta solución generada por Eclipse es asimétrica: solo en los métodos `add` y `remove` se hace el doble enganche, mientras que en los métodos `set` solo se realiza el enganche simple. Por lo tanto, en este tipo de soluciones es muy importante ser consciente de ello y utilizar preferiblemente el método `add` o `remove`.

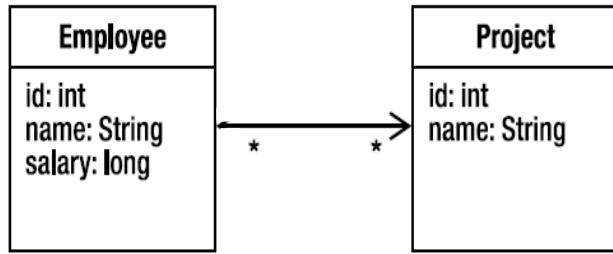
En todo caso, si no se opta literalmente, por ninguna de estas soluciones previas, es fundamental **trabajar sobre ambos lados explícitamente** (sobre la colección con los métodos `add` o `remove` de la propia API de `java.util` y sobre la referencia a uno, con su método `set`) para generar los enganches en **ambos sentidos**. **En caso contrario los objetos no estarán en un estado correcto.**

3.7 Many to many (muchos a muchos) – Unidireccional con Join Table

Un objeto está relacionado con un conjunto de objetos del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado con un conjunto de objetos del primer conjunto. **Solo se puede navegar desde el objeto del primer conjunto al del segundo.**

En el Dibujo 7, podemos ver la correspondencia con el diagrama de clases.





Dibujo 7: Ejemplo de Many to many - unidireccional con Join Table. Figura extraída de [Keith & Schincariol, 2009]

En este caso solo se añade anotación `@ManyToMany` en el lado **seleccionado como propietario** y desde donde se permite la navegación.

En el Código 9, se puede ver resuelto el ejemplo:

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    @JoinTable(name = "EMP_PROJ",
               joinColumns = @JoinColumn(name="EMP_ID"),
               inverseJoinColumns = @JoinColumn(name="PROJ_ID"))
    private Set<Project> projects;
    // ...
}

@Entity
public class Project {
    @Id private int id;
    private String name;
    // Remove (without navigation)
    // private Set<Employee> employees;
    // ...
}

```

Código 9: Ejemplo Many to many – Unidireccional con Join Table

En la entidad **seleccionada como no-proprietaria** se elimina toda información (atributos) vinculados a la entidad **proprietaria**. Incluyendo la eliminación de `mappedBy`.

Nota avanzada: en los ejemplos de `@ManyToMany` se utilizará `Set` en lugar de `List` por mejorar el rendimiento minimizando el número de operaciones SQL generadas en algunos frameworks como Hibernate.

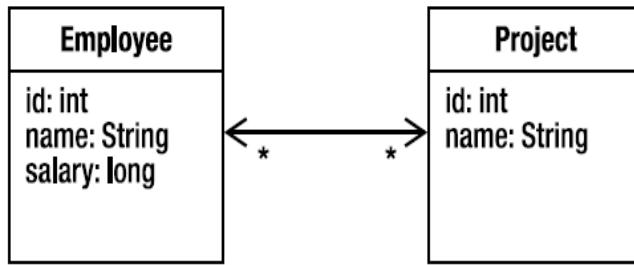
3.8 Many to many (muchos a muchos) – Bidireccional con Join Table

Un objeto está relacionado con un conjunto de objetos del segundo conjunto. Respectivamente, cada uno de estos objetos del segundo conjunto, está relacionado con un conjunto de objetos del primer conjunto.

Se permite la navegación en ambos sentidos

En el Dibujo 8, podemos ver la correspondencia con el diagrama de clases.





Dibujo 8: Ejemplo de Many to many - bidireccional con Join Table. Figura extraída de [Keith & Schincariol, 2009]

En JPA se resuelve con el uso obligatorio de JOIN_TABLE:

- Anotando a **nivel lógico** con `@ManyToMany`, en **ambas entidades**.
- No hay clave foránea en las tablas (están en la JOIN TABLE), por lo que **se selecciona un propietario**.
 - La entidad **proprietaria** incluye la información física de la JOIN TABLE con la anotación `@JoinTable`, nombre de tabla y sus correspondiente columnas claves foráneas (en directa y en inversa).
 - La entidad **no-proprietaria**. Incluye el elemento a **nivel lógico** `mappedBy`.

En el Código 10, se puede ver resuelto el ejemplo:

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    @JoinTable(name = "EMP_PROJ", joinColumns = @JoinColumn(name="EMP_ID"),
    inverseJoinColumns = @JoinColumn(name="PROJ_ID"))
    private Set<Project> projects;
    // ...
}

@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Set<Employee> employees;
    // ...
}
  
```

Código 10: Ejemplo Many to many – Bidireccional con Join Table

Si no se especifican nombres de tabla y columnas para la JOIN TABLE, se toman nombres por defecto (recordad “convención sobre configuración”). En el ejemplo se tomaría como nombre de la tabla la concatenación de las dos, EMPLOYEE_PROJECT y como columnas EMPLOYEE_ID y PROJECT_ID.

Al igual que **se ha comentado** en las `@OneToMany/@ManyToOne` bidireccionales, es MUY IMPORTANTE realizar el doble enganche entre ambas entidades, reutilizando las estrategias ya vistas previamente, añadiendo y completando los correspondientes métodos `add` y `remove`.

Por otro lado, si la JOIN_TABLE **contiene campos/atributos adicionales** (no solo las claves foráneas) se debe **crear una entidad adicional**, que incluya dichos campos/atributos. En este caso se establecen relaciones `@ManyToOne` a las tablas referenciadas, según la navegabilidad y las respectivas `@OneToMany` en



las tablas hacia la entidad originada por la JOIN_TABLE. Además se generará una clave compuesta derivada para la nueva entidad (el concepto de clave compuesta derivada se explica posteriormente en el apartado 6. Claves derivadas).

3.9 Orden en las asociaciones a varios

Si se quiere indicar el orden en que se recuperan los datos de las colecciones, en las asociaciones a varios, se utiliza la anotación @OrderBy. Se pueden indicar varios campos/columnas separados por comas, e indicar el orden ascendente (ASC) o descendente (DESC). Si no se indicar atributos, se asume ordenación por clave primaria.

```
@ManyToMany  
@JoinTable(name="BookAuthor",  
           joinColumns={@JoinColumn(name="bookId", referencedColumnName="id")},  
           inverseJoinColumns={@JoinColumn(name="authorId", referencedColumnName="id")})  
@OrderBy(value = "lastName ASC")  
private Set<Author> authors = new HashSet<Author>();
```

Código 11: Ejemplo del uso de @OrderBy

El resultado es que en la SQL generada se concatena la correspondiente cláusula ORDER BY.

4. Herencia

Un problema no trivial a la hora de abordar el *impedance mismatch* es la transformación de relaciones IS_A a jerarquías de herencia.

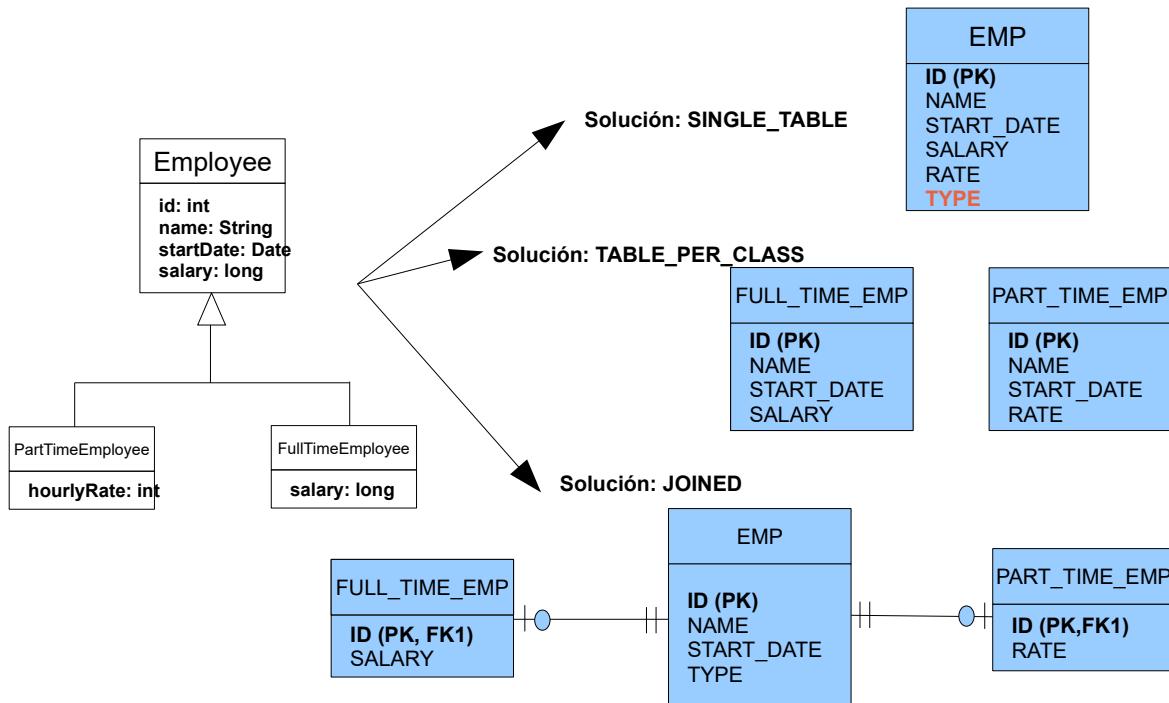
Este problema se complica más si tenemos en cuenta que en la jerarquía de herencia, las clases pueden ser abstractas o concretas, introduciendo una gran variante de casos. Si además añadimos la posibilidad de que sean o no entidades JPA, el problema se complica más, teniendo las siguientes variantes:

1. **Entidades JPA correspondientes a clases abstractas:** no pueden instanciarse, cumpliendo la norma general ya conocida en programación orientada a objetos, pero puede consultarse su valor a través de instancias de las subclases concretas.
2. **“Superclases mapeadas”¹:** las entidades JPA pueden heredar de **clases (abstractas o concretas) con estado persistente e información de mapeo, pero que NO son en sí entidades**. En estos casos se utiliza sobre la clase la **anotación @MappedSuperClass** en lugar de @Entity. No pueden ser consultadas, ni gestionadas, por el gestor de entidades del framework de persistencia, ni establecer relaciones y no tienen correspondencia con tablas.
3. **Superclases NO-entidad: las entidades JPA heredan de clases “normales” (no entidades, ni superclases mapeadas)**, pero **NO hay estado persistente a heredar** en estos casos. Al igual que antes, no pueden ser consultadas ni gestionadas por el gestor de entidades del framework de persistencia, ni establecer relaciones y no tienen correspondencia con tablas.

Con estas reglas en mente, se va a abordar a continuación la resolución del problema inicialmente planteado. Para ello se tomará el problema de partida, con el ejemplo mostrado en el Dibujo 9:

¹ Nos tomamos la licencia de hacer una traducción literal de la anotación @MappedSuperClass entendiendo que facilita la comprensión del caso.





Dibujo 9: Ejemplo de impedance mismatch entre jerarquía de herencia y tablas

De forma similar a como se abordó el uso de estrategias de generación de identificadores únicos, en JPA existen las correspondientes estrategias para resolver el mapeo de la jerarquía de herencia. En concreto se añadirá una anotación `@Inheritance` a la clase raíz, con un atributo `strategy` cuyo valor enumerado `InheritanceType` puede tomar tres posibles valores:

- **SINGLE_TABLE**: una sola tabla para la jerarquía.
- **TABLE_PER_CLASS**: una tabla por clase concreta.
- **JOINED**: tabla por clase (usando una estrategia de *join*).

4.1 Single table

En esta estrategia **toda la información de la jerarquía se almacena en una única tabla**. Para distinguir si la fila almacenada se corresponde a una u otra clase, se utiliza una columna con el discriminante (marcada con la anotación `@DiscriminatorColumn`).

La anotación `@DiscriminatorColumn` tiene como elementos adicionales, el nombre (`name`) de la columna (por defecto `DTYPE`), tipo de la columna (`discriminatorType`) (por defecto `DiscriminatorType.STRING`), información adicional para su definición (`columnDefinition`) y longitud (`length`).

El tipo de datos para el discriminante suele ser o `STRING`, `CHAR` o `INTEGER` (de más a menos habitual).

Posteriormente, en cada descendiente de tipo “clase concreta”, se utiliza la anotación `@DiscriminatorValue` para fijar el valor correspondiente a cada tipo. Si no se especifica, se utiliza el nombre de la entidad, por defecto.

Debemos señalar que en esta solución las columnas no comunes contendrán valores nulos.

A continuación, en el Código 12, se puede ver resuelto el caso:



```

@Entity
@Table(name="EMP")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
// in root class
@DiscriminatorColumn(name="EMP_TYPE") // in root class
public abstract class Employee { ... }

@Entity // default discriminator value "ContractEmployee"
public class ContractEmployee extends Employee { ... }

@MappedSuperclass // warning: using a mapped superclass...
public abstract class CompanyEmployee extends Employee { ... }

@Entity
@DiscriminatorValue("FTEmp")
public class FullTimeEmployee extends CompanyEmployee { ... }

@Entity(name="PTEmp") // default discriminator value "PTEmp"
public class PartTimeEmployee extends CompanyEmployee { ... }

```

Código 12: Herencia resuelta con Single Table

4.2 Table per class

En esta estrategia la información de la jerarquía se almacena en una **tabla por cada clase concreta**. **Cada tabla contiene la información común junto con la particular de cada caso**. Esta solución tiene pobre soporte para las relaciones polimórficas, puesto que requiere operaciones de UNION y JOIN con consultas SQL independientes para cubrir la jerarquía por completo. Además es **opcional** que el proveedor incluya o soporte esta estrategia.

Cada clase concreta pasa a ser una entidad persistente en JPA con su correspondiente tabla en la base de datos, como se muestra en el Dibujo 10.

CONTRACT_EMP		FT_EMP		PT_EMP	
PK	ID	PK	ID	PK	ID
	FULLNAME S_DATE D_RATE TERM		NAME S_DATE VACATION SALARY PENSION MANAGER		NAME S_DATE VACATION H_RATE MGR
		FK1		FK1	

Dibujo 10: Tablas correspondientes a Table per class. Figura extraída de [Keith & Schincariol, 2009]

Suponiendo que en el ejemplo previo, la clase `Employee` es abstracta, e introducimos una clase concreta `ContractEmployee` que hereda de ella, y una clase abstracta `CompanyEmployee` superclase de `FullTimeEmployee` y `PartialEmployee`, a continuación, en el Código 13, se puede ver resuelto el caso.



```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
// Without table...
public abstract class Employee {
    @Id private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}

@Entity
@Table(name="CONTRACT_EMP")
@AttributeOverrides({
    @AttributeOverride(name="name", column=@Column(name="FULLNAME")),
    @AttributeOverride(name="startDate", column=@Column(name="SDATE"))
})
public class ContractEmployee extends Employee {
    @Column(name="D_RATE")
    private int dailyRate;
    private int term;
    // ...
}

@MappedSuperclass
// Without table...
public abstract class CompanyEmployee extends Employee {
    private int vacation;
    @ManyToOne
    private Employee manager;
    // ...
}

@Entity
@Table(name="FT_EMP")
public class FullTimeEmployee extends CompanyEmployee {
    private long salary;
    @Column(name="PENSION")
    private long pensionContribution;
    // ...
}

@Entity
@Table(name="PT_EMP")
@AssociationOverride(name="manager", joinColumns=@JoinColumn(name="MGR"))
// Override a relationship (not an attribute) with new join column...
public class PartTimeEmployee extends CompanyEmployee {
    @Column(name="H_RATE")
    private float hourlyRate;
    // ...
}
```

Código 13: Herencia resuelta con Table per class

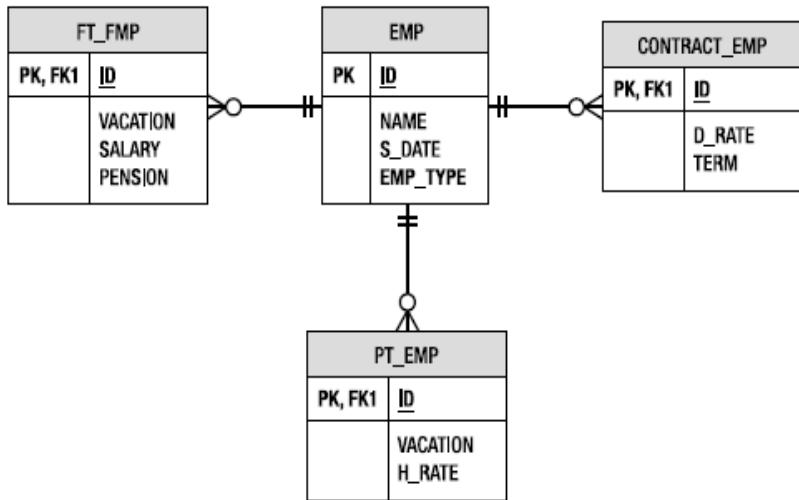
4.3 Joined

En esta estrategia se tiene **una tabla por cada clase**. **Cada tabla sólo contiene la información específica de cada caso**. Aunque da buen soporte polimórfico, requiere de uno o varios *joins* en la instanciación de las las subclases entidad. Igualmente las consultas sobre la jerarquía completa pueden requerir de varios joins.



El proveedor puede requerir además de una columna discriminante en la entidad raíz, por motivos de portabilidad (al igual que antes, usando `@DiscriminatorColumn` y `@DiscriminatorValue`).

Partiendo de un conjunto de tablas como el mostrado en el Dibujo 11:



Dibujo 11: Tablas correspondientes a Joined. Figura extraída de [Keith & Schincariol, 2009]

A continuación, en el Código 14, se puede ver resuelto el caso:

```

@Entity
@Table(name="EMP")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE", discriminatorType=DiscriminatorType.INTEGER)
public abstract class Employee { ... }

@Entity
@Table(name="CONTRACT_EMP")
@DiscriminatorValue("1")
public class ContractEmployee extends Employee { ... }

@MappedSuperclass // without table...
public abstract class CompanyEmployee extends Employee { ... }

@Entity
@Table(name="FT_EMP")
@DiscriminatorValue("2")
public class FullTimeEmployee extends CompanyEmployee { ... }

@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("3")
public class PartTimeEmployee extends CompanyEmployee { ... }
  
```

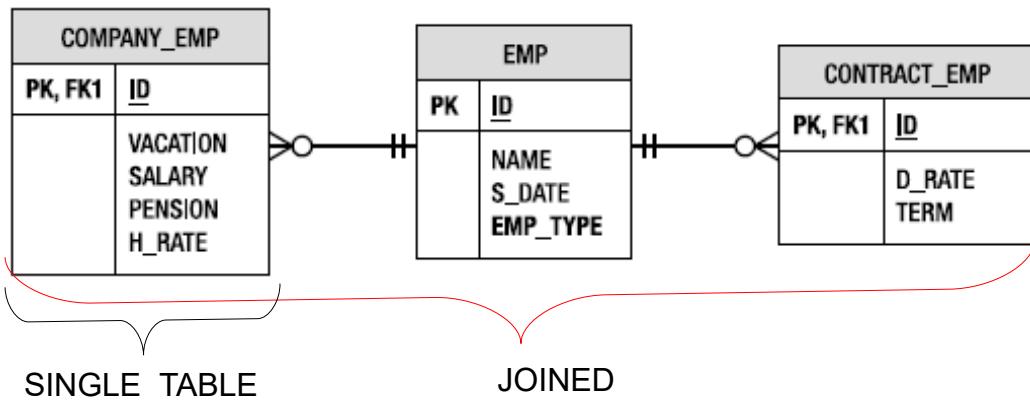
Código 14: Herencia resuelta con Joined



4.4 Mixed

No se trata en sí de una nueva estrategia, sino de la mezcla o combinación de varias. Esto permite el uso combinado de varias estrategias sobre la misma jerarquía. El problema fundamental es que esta solución **no está incluida en la especificación JPA** y podría no estar soportada por el proveedor. Por lo tanto evitaremos su uso.

Tomando como ejemplo el siguiente diagrama de clases, en el Dibujo 12.



Dibujo 12: Aplicando una estrategia Mixed. Figura extraída de [Keith & Schincariol, 2009]

Se resuelve el ejemplo, aplicando una estrategia de tabla única (SINGLE_TABLE) sobre COMPANY_EMP, que unifica los datos de FullTimeEmployee y PartialEmployee y de JOINED sobre las tres tablas mostradas.

A continuación, en el Código 15, se puede ver resuelto el caso, combinando ambas estrategias:



```

@Entity
@Table(name="EMP")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE", discriminatorType=DiscriminatorType.INTEGER)
public abstract class Employee {
    @Id private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}

@Entity
@Table(name="CONTRACT_EMP")
@DiscriminatorValue("1")
public class ContractEmployee
    extends Employee {
    @Column(name="D_RATE")
    private int dailyRate;
    private int term;
    // ...
}

@Entity
@Table(name="COMPANY_EMP")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
// using the discriminator of table EMP
public abstract class CompanyEmployee
    extends Employee {
    private int vacation;
    // ...
}

@Entity
@DiscriminatorValue("2")
public class FullTimeEmployee
    extends CompanyEmployee {
    private long salary;
    @Column(name="PENSION")
    private long pensionContribution;
    // ...
}

@Entity
@DiscriminatorValue("3")
public class PartTimeEmployee
    extends CompanyEmployee {
    @Column(name="H_RATE")
    private float hourlyRate;
    // ...
}

```

Código 15: Herencia resuelta con mixed (joined & single table)



5. Operaciones en relaciones

5.1 Carga de datos en relaciones

Como ya se comentó previamente, el *framework* de persistencia gestiona la carga de los valores en memoria de dos formas: perezosa (`LAZY`) o ansiosa (`EAGER`). Esto cobra mayor importancia en el momento que hemos establecido **relaciones entre las entidades**, y cargar los datos de una entidad puede acarrear cargar todos los datos de las entidades relacionadas con ella (e.g., en relaciones a varios con `@OneToMany`).

En concreto, el atributo `fetch` indica **cuándo** se carga el dato (atributo, entidad o colección): bien de modo ansioso al cargar la entidad raíz (`FetchType.EAGER`) o posteriormente de forma perezosa (`FetchType.LAZY`), al acceder a través de dicha entidad raíz a datos relacionados.

Al igual que con los atributos y propiedades, se puede indicar como elemento de la anotación en la relación el tipo de carga que se quiere. Aunque es potestad del proveedor implementar el modo perezoso (podría realizar siempre el modo ansioso).

Sin embargo el **cómo se cargan los datos** es otra cuestión, independientemente del momento. Básicamente se trabaja en JPA con dos modos:

- **Estrategia JOIN:** aplicada inherentemente con EAGER al usar búsquedas por clave primaria, con el método `find` del `EntityManager`. Internamente se resuelve con una `SELECT` con `LEFT OUTER JOIN` entre tablas, generando una **única consulta**.
- **Estrategia SELECT:** aplicada con EAGER o LAZY, en consultas con JPQL/Criteria API (siguiente sección). Sin embargo esto acaba generando consultas adicionales y el problema denominado `N+1`. Con EAGER las genera **inmediatamente** (respetando el `fetch` indicado en el modelo estático) mientras que con LAZY cuando se accede **posteriormente**.

Por ejemplo, si queremos que una vez cargado un empleado, no se carguen las correspondientes plazas de garaje, hasta que realmente se acceda a los datos, se utilizaría `FetchType.LAZY` en el siguiente Código 16.

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    // ...
}
```

Código 16: Carga perezosa de datos en una relación

En JPA la carga por defecto es **dependiente** de la **multiplicidad/cardinalidad** de la entidad destino a la que se apunta, en concreto:

- Para relaciones To-One el modo por defecto es `EAGER`.
 - Tanto para `@OneToOne` como `@ManyToOne`.
 - Carga ansiosamente una instancia/fila relacionada.
- Para relaciones To-Many el modo por defecto es `LAZY`
 - Tanto para `@OneToMany` como `@ManyToMany`.
 - Carga perezosamente una colección de instancias/filas cuando sea solo estrictamente necesario (se accede explícitamente).



Aunque dichos modos se puede “redefinir” con atributos en las anotaciones correspondientes, es conveniente conocer su comportamiento por defecto, más cuando influyen en el número de consultas generadas. En concreto, el valor EAGER por defecto para relaciones a uno, es criticado, y en la bibliografía se suele recomendar redefinir dicho valor a LAZY.

En temas posteriores se revisará con mayor detalle su implicación en el número de consultas generadas por parte del *framework*. Y las soluciones existentes para reducir el número de consultas utilizando estrategias de JOIN/FETCH en consultas JPQL/Criteria API o bien con los grafos de entidades de forma dinámica.

Nota avanzada en Hibernate y EclipseLink: en estos *frameworks* se denomina estrategia JOIN a la generación de una consulta con JOIN (valor por defecto con EAGER al usar el `find` ligado al `EntityManager`) y estrategia SELECT (valor por defecto para consultas JPQL/Criteria API) a la generación de consultas adicionales. También se definen estrategias SUBSELECT y BATCH para recuperar valores con subconsultas y en modo *batch*. En Hibernate, en concreto, se utiliza una anotación `@Fetch` con atributos `FetchMode.SELECT`, `FetchMode.SUBSELECT` o `FetchMode.JOIN` y una anotación `@BatchSize`. En EclipseLink hay anotaciones `@JoinFetch` y `@BatchFetch`. Esto NO es parte del estándar JPA y queda fuera del ámbito de la asignatura.

5.2 Mantenimiento de relaciones

Como hemos visto, el programador indica y mantiene las relaciones entre entidades. Pero para facilitar el manejo de las entidades relacionadas se puede apoyar en mecanismos en cascada. Estos mecanismos se aplican automáticamente² sobre las entidades relacionadas, propagando las acciones adecuadas.

Para ello se añade un elemento `cascade` a la anotación de relación con los siguientes valores:

Operación	Descripción
ALL	Aplicar todas las operaciones a las entidades relacionadas. Equivalente a <code>cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}</code>
DETACH	Si el padre se desvincula del contexto de persistencia, la entidad relacionada también.
MERGE	Si la entidad padre es mezclada (se vuelcan los cambios de una entidad desvinculada) en el contexto de persistencia, la entidad relacionada también
PERSIST	Si la entidad padre se hace persistente en el contexto de persistencia, la entidad relacionada también
REFRESH	Si la entidad padre es refrescada en el contexto de persistencia, la entidad relacionada también
REMOVE	Si la entidad padre es eliminada en el contexto de persistencia, la entidad relacionada también. Debe ser utilizada sólo en <code>@OneToOne</code> y <code>@OneToMany</code> sobre la entidad “one”

Además en JPA 2.x `orphanRemoval="true"`, elimina entidades huérfanas en relaciones `@OneToOne` o `@OneToMany` si han sido desconectadas (e.g. poner valor a `null`, conectar con otra entidad, etc.) dejando objetos desconectados, que no serían borrados en cascada (y son huérfanos).

Por ejemplo, en el Código 17. se muestra un ejemplo de uso de estas opciones sobre las relaciones definidas en la entidad empleado.

² Lo indicado en este apartado, se revisará y aclarará, al explicar el concepto de gestor de entidades en la siguiente sección. Existen dependencias circulares entre ambos secciones, que dificultan explicar una, sin haber visto la otra.



```

@Entity
public class Employee {
// ...
    @OneToOne
    ParkingSpace parkingSpace;

    @OneToMany(mappedBy="employee", cascade=CascadeType.DELETE)
    Collection<Phone> phones;

    @OneToMany(fetch = FetchType.LAZY,
    cascade = {CascadeType.PERSIST,CascadeType.MERGE},
    mappedBy = "manager")
    List<Employee> employees;

    @ManyToOne
    Address address;
// ...
}

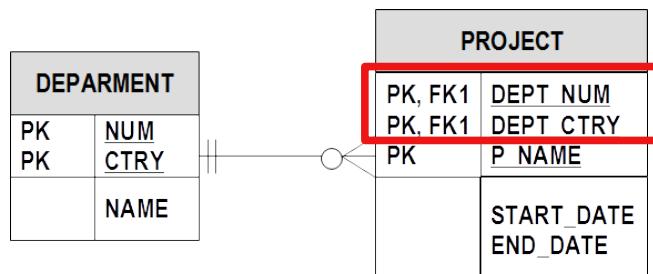
```

Código 17: Ejemplo de uso de opción cascade

6. Claves derivadas

Anteriormente se indicó que las claves primarias pueden ser simples o compuestas. En el caso de las claves compuestas, al utilizar relaciones, se puede dar el caso de que estén compuestas a su vez por la clave foránea de otra entidad padre. **En estos casos hablamos de claves derivadas**. Esto se produce típicamente en la dependencia en identificación hacia la no-propietaria en @ManyToOne o @OneToOne.

Por ejemplo, si tomamos departamentos y proyectos asociados, donde los proyectos se identifican por el departamento donde se realizan (ver Dibujo 13), se establece una clave primaria compuesta y derivada. Debemos tener en cuenta además que el departamento tiene una clave compuesta en el ejemplo.



Dibujo 13: Clave primaria compuesta y derivada.

A continuación se resuelve este ejemplo con las dos variantes ya vistas para claves compuestas, usando una clase de identificación o bien con un identificador embebido.

6.1 Solución con @IdClass

En el primer caso se construye una clase de identificación tanto para el proyecto (`ProjectId`) como para el departamento (`DeptId`). En la clase `ProjectId` se referencia al identificador de departamento (`DeptId`).



En el siguiente código se muestra el código asociado a las clases de identificación (ver Código 18)

```
public class ProjectId implements Serializable {
    private String name;
    private DeptId dept;
    public ProjectId() {}
    public ProjectId(DeptId deptId, String name) {
        this.dept = deptId;
        this.name = name;
    }
    // ...
}

public class DeptId implements Serializable {
    private int number;
    private String country;
    public DeptId() {}
    public DeptId (int number, String country) {
        this.number = number;
        this.country = country;
    }
    // ...
}
```

Código 18: Clases de identificación para clave primaria compuesta derivada

Por otro lado en las clases correspondientes de proyecto y departamento se utiliza la anotación `@IdClass` correspondiente, salvo que en el caso de proyecto **se utiliza sobre el atributo que modela la relación** (ver Código 19). Además se utilizan los `@JoinColumns` para resolver el mapeo, dado que hay clave primaria compuesta en `Department`.

En este caso la correspondencia no es uno a uno entre los atributos que forman la clave primaria y los atributos definidos en la clase de identificación, puesto que existe un desajuste entre `dept` de tipo `Department` en `Project`, y `dept` de tipo `DeptId` en `ProjectId`.

```
@Entity
@IdClass(ProjectId.class)
public class Project {
    @Id private String name;

    @Id
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="DEPT_NUM", referencedColumnName="NUM"),
        @JoinColumn(name="DEPT_CTY", referencedColumnName="COUNTRY") })
    private Department dept;
    // ...
}
```

Código 19: Clase proyecto con clave compuesta derivada utilizando clase de identificación

Nota importante: los `@IdClass` con clave derivada en Eclipse, generan error en compilación (debido a un bug en el plugin Eclipse Dali) por lo que no se utilizará esta opción en prácticas, utilizando siempre la solución equivalente con `@EmbeddedId`.

6.2 Solución con `@EmbeddedId`

En el segundo caso se construye un identificador embebido tanto para el proyecto (`ProjectId`) como para el departamento (`DeptId`). Al igual que antes en la clase `ProjectId` se referencia al identificador de departamento. En la clase `Project`, propietaria, se indica el atributo que representa la relación con `@MapsId`.



```

@Embeddable
public class ProjectId implements Serializable {
    @Column(name="P_NAME")
    private String name;
    @Embedded
    private DeptId dept;
    // ...
}

@Embeddable
public class DeptId implements Serializable {
    @Column(name="NUM")
    private int number;
    @Column(name="CTRY")
    private String country;
    // ...
}

```

Código 20: Clases embebidas de identificación para clave primaria compuesta derivada

En la clase `Project` se añade la correspondiente anotación `@MapsId` en la relación, indicando el nombre del atributo correspondiente en el tipo embebido `ProjectId` (ver Código 21). Es necesario también usar el mapping `@JoinColumns` dado que `Department` tiene una clave primaria compuesta, como en el caso anterior.

```

@Entity
public class Project {
    @EmbeddedId private ProjectId id;

    @MapsId("dept")
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="DEPT_NUM", referencedColumnName="NUM"),
        @JoinColumn(name="DEPT_CTRY", referencedColumnName="CTRY") })
    private Department department;
    // ...
}

@Entity
public class Department {
    @EmbeddedId private DeptId id;
    @OneToMany(mappedBy="department")
    private List<Project> projects;
    // ...
}

```

Código 21: Clases utilizando identificadores embebidos con clave primaria compuesta derivada

6.3 Clave primaria compartida

Como caso muy particular, podemos tener que la clave primaria no sea compuesta, sino simple pero definida por la entidad de la que se depende en una relación `@OneToOne`.

Supongamos el caso de una entidad `EmployeeHistory`, donde su clave primaria está formada solo por la clave foránea en la tabla de empleados, de la que depende. Una opción es definir el atributo que modela la relación, como clave primaria simple.



Si además se quiere tener acceso al valor de clave primaria, es necesario añadir el atributo y en la relación añadir la anotación `@MapsId`.

En el siguiente Código 22, se muestran ambas variantes.

```
@Entity
public class EmployeeHistory {
    // ...
    @Id
    @OneToOne
    @JoinColumn(name="EMP_ID")
    private Employee employee;
    // ...
}

@Entity
public class EmployeeHistory {
    // ...
    @Id
    int empId;

    @MapsId
    @OneToOne
    @JoinColumn(name="EMP_ID")
    private Employee employee;
    // ...
}
```

Código 22: Variantes para clave primaria compartida

6.4 Reglas generales

Concluyendo el apartado, se indican a continuación las reglas generales a cumplir en el uso de claves primarias compuestas derivadas.

- Una entidad dependiente puede tener múltiples entidades padres, por lo tanto el id. puede incluir varias claves foráneas.
- La entidad dependiente debe establecer las relaciones a sus entidades padre antes de invocar al método para hacerse persistente (`persist`).
- Si tiene múltiples atributos `@Id`, debe usar una `@IdClass` y debe haber el correspondiente atributo con el mismo nombre en la `@IdClass`.
- Los atributos `@Id` pueden ser tipos simple o tipo entidad, que es objetivo de una `@ManyToOne` o `@OneToOne`.
- Si un atributo `@Id` en una entidad es de tipo simple, entonces el tipo del atributo en la `@IdClass` debe ser el mismo tipo simple.
- Si un atributo `@Id` en una entidad es una relación, entonces el tipo del atributo en la `@IdClass` es del mismo tipo que la clave primaria de la entidad objetivo en la relación (ya sea tipo simple, `@IdClass` o un id.embebido)
- Si el identificador derivado es una clase de id.embebido (`@EmbeddedId`) entonces cada atributo de la clase de identificación, que representa la relación, debería referenciarse con `@MapsId` en el correspondiente atributo de la relación.

7. Clases embebidas (Revisión)

Como se vio en la lección previa, las clases embebidas pueden utilizarse para modelar asociaciones de composición, donde la existencia del objeto embebido está íntimamente ligado al objeto que lo contiene. Como ejemplo típico hemos visto el caso de las direcciones de los empleados.

Sin embargo surge ahora el problema de cómo compartir **clases embebidas (NO compartir instancias)**. El problema se produce porque puede que no exista siempre una coincidencia de nombres en el uso de la clase desde otras.



Por ejemplo, supongamos que tenemos dos tablas, `EMPLOYEE` y `COMPANY` donde cada una define los campos correspondientes a la dirección pero con distintos nombres (ver Dibujo 14).

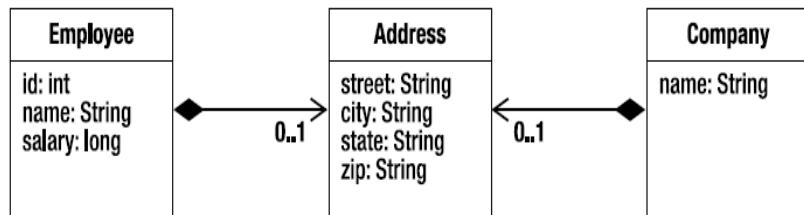
EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE

Dibujo 14: Tablas "compartiendo" la definición de dirección.

Figura extraída de [Keith & Schincariol, 2009]

Si modelamos esto en un diagrama de clases (ver Dibujo 15) vemos que existe un cierto desajuste entre los nombre utilizados en las tablas y el único nombre utilizado en la clase, cuando esta es compartida.



Dibujo 15: Modelo de clases compartiendo la clase `Address`. Figura extraída de [Keith & Schincariol, 2009]

Como solución, se plantea la redefinición de nombres utilizando anotaciones `@AttributeOverride` y `@Attribute` como se puede ver en el Error: no se encontró el origen de la referencia. Se deja pendiente completar la solución para la clase `Company`, que se resuelve de forma equivalente.

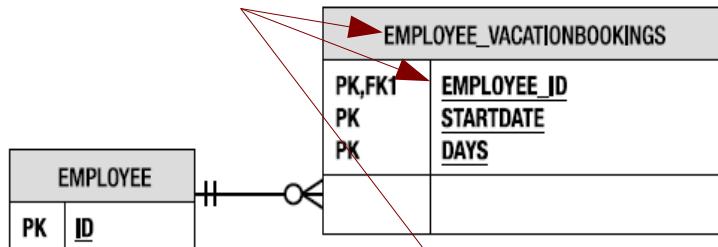
8. Uso de colecciones de elementos (Revisión)

En la práctica las **relaciones** definen asociaciones entre **entidades independientes**. Sin embargo nos podemos encontrar con colecciones de elementos que son **objetos dependientes de la entidad contenedora (en identificación) y solo accesibles a través de ella**. Se pueden implementar a través de colecciones de tipos embebidos o básicos.

Se utilizan solo interfaces de `java.util`, en concreto `Collection`, `List`, `Set` y `Map`, junto con la anotación `@ElementCollection`. (ver Dibujo 16 y Código 23).



Nombres por defecto



```

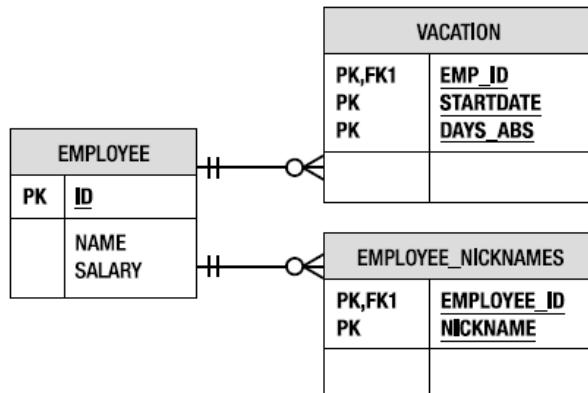
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // Using default collection tables
    @ElementCollection(targetClass=VacationEntry.class)
    private Collection vacationBookings;
    @ElementCollection
    private Set<String> nickNames;
    // ...
}

@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;
    @Column(name="DAYS")
    private int daysTaken;
    // ...
}

```

Código 23: Colecciones con nombres por defecto

Si además no se utilizan nombres por defecto (se prefiere “configurar” frente a la “convención”) es necesario utilizar `@CollectionTable` para **reconfigurar el mapeo de tablas y columnas** (ver Dibujo 17 y Código 24). El proveedor de persistencia asume que la combinación de atributos son únicos en combinación con las la clave foránea (JoinColumn(s))



Dibujo 17: Uso de colecciones sin nombres por defecto.
Figura extraída de [Keith & Schincariol, 2009]



```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass=VacationEntry.class)
    @CollectionTable(
        name="VACATION",
        joinColumns=@JoinColumn(name="EMP_ID"))
    @AttributeOverride(
        name="daysTaken",
        column=@Column(name="DAYS_ABS"))
    private Collection vacationBookings;
    @ElementCollection
    @Column(name="NICKNAME")
    private Set<String> nickNames;
    // ...
}
```

Código 24: Colecciones sin nombres por defecto

En este segundo ejemplo, dado que no se ha seguido la convención en la tabla VACATION, ni en sus campos, es necesario ajustar en el código los nombres a utilizar. No ocurre así con la tabla EMPLOYEE_NICKNAMES.

9. Anexo: Interfaces Java en java.util con JPA

Los anexos de estas secciones contienen material adicional que **NO entra como materia de evaluación**, pero se deja disponible para su consulta posterior, por si fuera de utilidad para los alumnos en otros contextos.

A la hora de utilizar colecciones en JPA se recomienda el uso de interfaces Java definidas en el paquete java.util. Dentro de los tipos recomendados están:

- **Collection**: no importa qué implementación concreta se conecte, y se usan los métodos generales
- **Set**: impide elementos duplicados y es muy simple de usar con menos generalidades
 - *Nota: ni Collection ni Set utilizan anotaciones JPA adicionales*
- **List**: cuando interesa recuperar las entidades en un cierto orden. Bien por el propio estado de las entidades en la lista o bien con una columna adicional (más intrusivo).
- **Map**: cuando se necesita asociar claves con valores, además con buenos tiempos de búsqueda sobre clave

Dado que **List** y **Map** sí que incluyen el uso de anotaciones JPA especiales, se describe a continuación su uso de manera más detallada.

9.1 List

Cuando se requiere ordenar los resultados, por comparación de un valor de la entidad, se utiliza la anotación **@OrderBy**. Si no se utiliza se devuelven los valores ordenados por clave primaria (en el caso de los embebidos en el orden que devuelva la base de datos).

Si se quiere ordenar explícitamente por algún atributo hay que indicarlo: en orden ascendente – por defecto – (ASC) o descendente (DESC). Es requisito que los atributos sean comparables.



Cambiar el orden de los elementos una vez cargados e memoria no tiene efecto con la relación, una lectura posterior volverá a devolverlos en el orden marcado por `@OrderBy`.

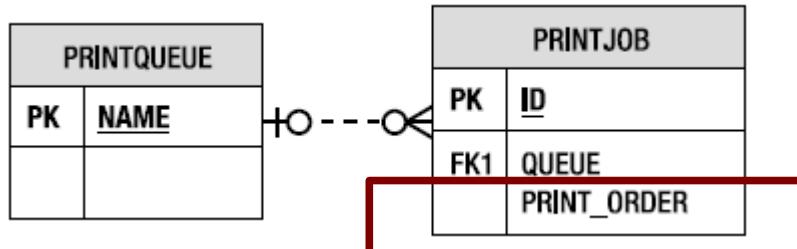
En el siguiente Código 25, se muestra un ejemplo de uso de `@OrderBy`.

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @OrderBy("name ASC, status DESC")
    private List<Employee> employees;
    // ...
}
```

Código 25: Ejemplo de uso de `@OrderBy` con `List`

Si por el contrario **no se puede ordenar por ninguno de los atributos** (no se puede aplicar `@OrderBy`) se puede añadir una columna adicional con el orden, transparente al usuario, `@OrderColumn`, de tipo entero. Dicho valor se actualiza sin interacción por parte de la aplicación (el proveedor de persistencia es responsable de actualizar el orden reflejando cualquier inserción, borrado o reordenación) pero no es una solución con buen rendimiento.

En el Dibujo 18, se muestra la solución de uso de una columna adicional para ordenaciones y en el Código 26, el mapeo correspondiente.



Dibujo 18: Uso de columna adicional para ordenaciones. Figura extraída de [Keith & Schincariol, 2009]

```
@Entity
public class PrintQueue {
    @Id private String name;
    // ...
    @OneToMany(mappedBy="queue")
    @OrderColumn(name="PRINT_ORDER") // Not use @OrderBy
    private List<PrintJob> jobs;
    // ...
}
```

Código 26: Ejemplo de columna artificial con `@OrderColumn` para ordenaciones en `List`

9.2 Map

El uso de `Map` es complicado puesto que se permite el uso de entidades, embebidos y tipos básicos para la definición de la clave-valor utilizada (a partir de JPA 2.x). Esto permite un gran número de posibles combinaciones.

Sin embargo hay una restricción y es que se deben usar **claves únicas**, lo que obliga a que los métodos `equals` y `hashCode` estén correctamente redefinidos.



A continuación se muestra un primer ejemplo con claves de tipo básico y colecciones, en el Error: no se encontró el origen de la referencia. Se indica la columna que cumple el papel de clave (@MapKeyColumn) y la columna que juega el papel de valor (@Column).

Suponemos que la estructura de tablas es la mostrada en el Error: no se encontró el origen de la referencia.

En lugar de utilizar tipos básicos, se puede mejorar la propuesta utilizando tipos enumerados. Esta variante obliga al uso de la anotación @MapKeyEnumerated. Podemos ver un ejemplo en el Código 27. Existe una versión similar de anotación @MapKeyTemporal para el uso de datos de tipo temporal.

```
public enum PhoneType { Home, Mobile, Work }

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyEnumerated(EnumType.STRING) // using the text
    @MapKeyColumn(name="PHONE_TYPE")
    @Column(name="PHONE_NUM")
    private Map<PhoneType, String> phoneNumbers;
    // ...
}
```

Código 27: Colección con tipo enumerado en clave de un Map

También se pueden usar claves con tipo básico para las relaciones @OneToMany. Por ejemplo, supongamos que los empleados trabajan en una ubicación con clave CUB_ID en la base de datos, podríamos definir la clase de la siguiente forma (ver Código 28).

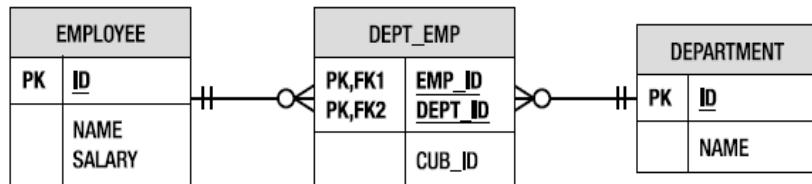
```
@Entity
public class Department {
    @Id private int id;
    @OneToMany(mappedBy="department")
    @MapKeyColumn(name="CUB_ID")
    private Map<String, Employee> employeesByCubicle;
    // ...
}
```

Código 28: Relación con clave tipo básico en un Map

Por otro lado también se pueden utilizar tipos básicos en relaciones @ManyToMany. Suponiendo la estructura de tablas mostrada en Dibujo 19, el código correspondiente utiliza de nuevo la anotación @MapKeyColumn (ver Código 29), donde cada empleado puede trabajar en varios departamentos.



Podemos encontrar el caso de que la clave esté formada por atributos de la entidad en una relación



Dibujo 19: Tablas para implementación de @ManyToMany con tipos básicos en un Map. Figura extraída de [Keith & Schincariol, 2009]

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @ManyToMany
    @JoinTable(name="DEPT_EMP",
    joinColumns=@JoinColumn(name="DEPT_ID"),
    inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @MapKeyColumn(name="CUB_ID")
    private Map<String, Employee> employeesByCubicle;
    // ...
}
```

Código 29: Relación con clave tipo básico en @ManyToMany con un Map

@OneToMany. Se usa la anotación @MapKey sobre el atributo de la entidad destino en la relación. Si no se especifica el name se utiliza la PK, si se especifica, debe ser un valor único en el contexto de la relación (ver Código 30).

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @MapKey(name="id")
    // id is key (@id) in Employee, type Integer or int
    private Map<Integer, Employee> employees;
    // ...
}
```

Código 30: Relación con clave definida en la entidad relacionada en @OneToMany con un Map

También se pueden definir claves con embebidos, aunque se debe evitar su uso, dado que los embebidos no tienen identidad (hay que combinar la identidad con la entidad fuerte) (ver Código 31).



```

@Entity
public class Employee {
    @Id private int id;
    @Column(name="F_NAME")
    private String firstName;
    @Column(name="L_NAME")
    private String lastName;
    private long salary;
    // ...
}

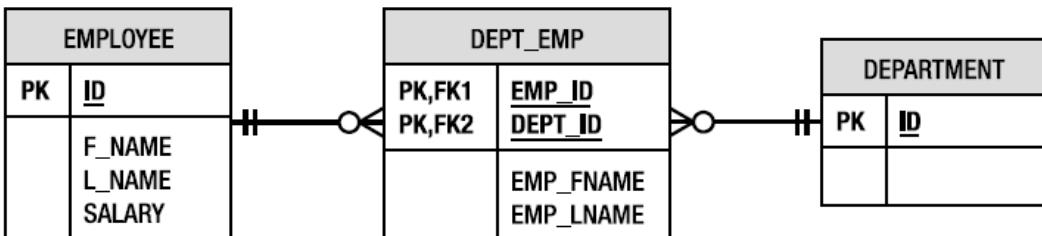
@Entity
public class Department {
    // ...
    @OneToOne(mappedBy="department")
    private Map<EmployeeName, Employee> employees;
    // ...
}

@Embeddable
public class EmployeeName {
    @Column(name="F_NAME", insertable=false, updatable=false)
    private String first_Name;
    @Column(name="L_NAME", insertable=false, updatable=false)
    private String last_Name;
    // ...
}

```

Código 31: Uso de embebidos para la clave en `@OneToMany` con un Map

En el siguiente ejemplo se muestra otra solución con embebidos, suponiendo una relación `@ManyToMany` como se muestra en Dibujo 20, resuelta con el Código 32.



Dibujo 20: Tablas para implementación con embebido en la clave para `@ManyToMany` con un Map. Figura extraída de [Keith & Schincariol, 2009]



```

@Entity
public class Employee {
    @Id private int id;
    @Embedded
    private EmployeeName name;
    private long salary;
    // ...
}

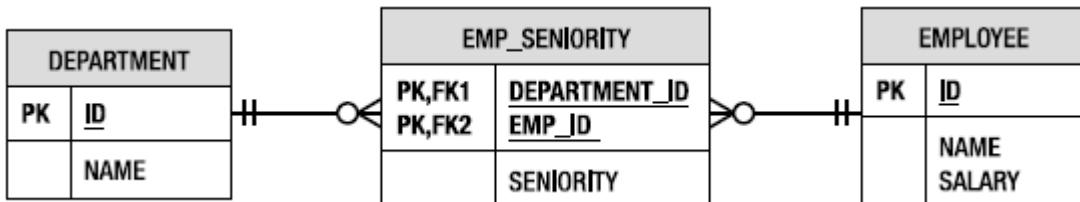
@Entity
public class Department {
    @Id private int id;
    @ManyToMany
    @JoinTable(name="DEPT_EMP",
               joinColumns=@JoinColumn(name="DEPT_ID"),
               inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @AttributeOverrides({
        @AttributeOverride(
            name="first_Name",
            column=@Column(name="EMP_FNAME")),
        @AttributeOverride(
            name="last_Name",
            column=@Column(name="EMP_LNAME"))
    })
    private Map<EmployeeName, Employee> employees;
    // ...
}

@Embeddable
public class EmployeeName {
    @Column(name="F_NAME")
    private String first_Name;
    @Column(name="L_NAME")
    private String last_Name;
    // ...
}

```

Código 32: Uso de embedidos para la clave en @ManyToMany con un Map

Finalmente se plantea el uso de entidades como claves. Aunque aparentemente costoso, está bien gestionado por JPA. En el Dibujo 21, se muestra la estructura de tablas y su posterior implementación en el Código 33.



Dibujo 21: Tablas para implementación con entidad en la clave con un Map. Figura extraída de [Keith & Schincariol, 2009]



```
@Entity
public class Department {
    @Id private int id;
    private String name;
    // ...
    @ElementCollection
    @CollectionTable(name="EMP_SENIORITY")
    @MapKeyJoinColumn(name="EMP_ID")
    @Column(name="SENIORITY")
    private Map<Employee, Integer> seniorities;
    // ...
}
```

Código 33: Uso de entidad para la clave con un Map

10. Resumen

En esta sección se han planteado el mapeo básico de relaciones con JPA. Desde el estudio de las relaciones fundamentales a casos más avanzados como la herencia. A partir de estos casos se revisan conceptos ya vistos en la operación con relaciones, claves derivadas, uso de clases embebidas, para finalizar con el uso de colecciones de elementos y un anexo particular del uso de estructuras de datos Java.

11. Glosario

E-R: modelo Entidad-Relación.

12. Bibliografía

[Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol. **Pro JPA 2. A definitive guide to mastering the Java Persistence API (2013)** Apress. 2nd edition.

[Keith et al., 2018] Mike Keith, Merrick Schincariol, Massimo Nardone. **Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence APIs.** (2018) Apress. 3rd edition.

[Oracle, 2017] The Java EE 8 Tutorial (2017). Part VIII. Persistence. Disponible en <https://javaee.github.io/tutorial/toc.html>

[Oracle, 2014] The Java EE 7 Tutorial (2014). Part VIII. Persistence. Disponible en <http://docs.oracle.com/javaee/7/tutorial/>



13. Recursos

Bibliografía complementaria:

Java Persistence. (2021, September 30). *Wikibooks, The Free Textbook Project*. Retrieved 08:08, April 9, 2022 from https://en.wikibooks.org/w/index.php?title=Java_Persistence&oldid=3992576.



Licencia

Autores: Raúl Marticorena & Mario Martínez & Pablo García

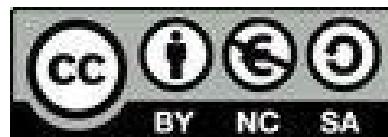
Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2022



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <https://creativecommons.org/licenses/by-nc-sa/4.0/>





Grado en Ingeniería Informática

APLICACIONES DE BASES DE DATOS

TEMA 5

Sección 4. JPA: Consultas

Docentes:

Raúl Marticorena
Jesús Maudes
Ana Serrano
Óscar Pérez

Índice de contenidos

1. INTRODUCCIÓN.....	3
2. OBJETIVOS.....	3
3. PROBLEMA INICIAL.....	3
4. JPQL.....	3
4.1 Características.....	4
4.2 Subconsultas.....	7
4.3 Definición de consultas.....	8
4.4 Retorno de consultas.....	11
4.5 Soporte de paginación.....	12
4.6 Consultas con herencia.....	13
4.7 Bulk update y delete.....	14
4.8 Consultas nativas.....	16
5. GRAFOS DE ENTIDADES.....	18
5.1 Planteamiento del problema.....	18
5.2 Definición y uso de grafos de entidades.....	18
6. AUDITORÍA Y OPTIMIZACIÓN.....	21
7. ANEXO 1. CRITERIA API.....	23
7.1 Metaclasses.....	24
7.2 Construcción de consultas.....	25
7.3 Expresiones.....	26
7.4 Ejecución de consultas.....	28
7.5 Subconsultas.....	28
7.6 Comparativa JPQL vs. Criteria API.....	30
8. ANEXO 2. USOS AVANZADOS.....	30
8.1 Uso de procedimientos almacenados desde JPA.....	30
8.2 Uso avanzado de Criteria API para actualizaciones y borrados.....	32
9. RESUMEN.....	32
10. GLOSARIO.....	33
11. BIBLIOGRAFÍA.....	33
12. RECURSOS.....	33



1. Introducción

En esta sección se introduce el concepto de consultas en JPA. Frente a la solución bien conocida del uso de sentencias SELECT con SQL en bases de datos relaciones, surge la problemática de resolver de nuevo el desajuste con el modelo de objetos y la navegación a través de las asociaciones entre los mismos.

Como soluciones: surge una variante de SQL para JPA denominada JPQL, y una API completa modelando consultas y permitiendo su construcción, denominada Criteria API.

2. Objetivos

- Conocer el problema de impedancia entre consultas con SQL y consultas con objetos.
- Describir y utilizar la definición de consultas con JPQL.
- Desglosar la Criteria API para la construcción de consultas en JPA.

3. Problema inicial

Para las labores de análisis e inteligencia del negocio, es fundamental tener la capacidad de consulta de la información en nuestras aplicaciones (ya sea de la BD o del modelo de objetos). Sin embargo, el problema de la impedancia entre ambos modelos, se acusa de manera particular en este apartado.

Ante cuestiones como: *¿cómo consultar la información de los objetos cargados en memoria?* Surge una doble respuesta: *¿seguimos pensando en navegación de objetos o seguimos pensando en proyecciones y JOINs de tablas?*

La solución que se ha dado, es abstraer de nuevo definiendo "nuevos" lenguajes de consulta, adaptando SQL al modelo de objetos (surgiendo JPQL), o utilizando metaclasses que permitan construir consultas y navegar sobre los objetos con Criteria API.

Estas opciones ofrecen una ventaja (frente a la desventaja de tener que aprender una nueva variante) y es la **independencia del SQL generado por el proveedor de persistencia**, de forma transparente a la aplicación y reduciendo dependencias concretas.

4. JPQL

JPQL¹ se corresponde con las siglas de Java Persistence Query Language. Su propio nombre es descriptivo, se trata de un nuevo lenguaje de consulta para usarse con entidades persistentes Java. Tomando una definición más formal diríamos que es:

Lenguaje de consulta independiente de la base de datos que opera en el modelo lógico de entidades en contraposición al modelo físico de datos.

Nos encontramos con una versión mejorada de EJB QL, lenguaje de consulta inicialmente ideado en la especificación de *Enterprise JavaBeans* (EJB) para su uso particular con entidades persistentes (*Entity Beans*).

¹ En la bibliografía nos encontraremos las siglas escritas como JPQL o JPQL indistintamente. Por analogía con SQL de aquí en adelante en estos apuntes se utilizará preferentemente JPQL.



4.1 Características

Sintaxis similar y familiar a SQL, reduciendo el salto y la curva de aprendizaje, pero también con fuertes influencias de la notación orientada a objetos (i.e. uso de notación de punto, etc.)

En el siguiente Código 1 podemos ver un par de sentencias JPQL.

```
SELECT e  
FROM Employee e  
  
SELECT e.name  
FROM Employee e
```

Código 1: Ejemplos de consultas JPQL sobre una entidad Employee

En el primer caso realizamos una proyección sobre la entidad `Employee` (correspondiente a la tabla `EMPLOYEE`), recuperando el conjunto de objetos de tipo `Employee` correspondiente al conjunto de tuplas o filas de la tabla.

En la segunda consulta se proyecta solo sobre la columna nombre de la entidad recuperando el conjunto de nombres de sus filas. Debemos remarcar el uso de **variables de entidad** (`e` en el ejemplo) y el uso de la **notación del punto** para seleccionar propiedades concretas de la entidad (`e.name`).

Sin embargo la sintaxis utilizada hereda básicamente la estructura de una SELECT en SQL, indicando la proyección y el origen del que extraer los datos.

Como podemos observar, se permite también la navegación sobre los objetos, aunque estos no estén explícitamente en la cláusula FROM. En el siguiente ejemplo (ver Código 2), podemos ver como se recupera el conjunto de departamentos (entidades de tipo `Department`) asociados a los empleados, navegando con la notación del punto a través de su relación (`@ManyToOne`). Estamos navegando sobre el modelo de objetos a través de la sentencia JPQL.

```
SELECT e.department  
FROM Employee e
```

Código 2: Navegación sobre objetos no explícitamente indicados en la cláusula FROM

Siguiendo el paralelismo con la SELECT en SQL, se pueden completar las consultas con su correspondientes cláusulas WHERE y uso de operadores como IN, LIKE, BETWEEN, funciones de cadena y subconsultas.

En el Código 3, tenemos una nueva consulta sobre los empleados, filtrando las entidades resultante por aquellas cuyo departamento es 'NA42' y el estado de su dirección está en el conjunto {'NY', 'CA'}. Es importante señalar de nuevo la navegación en el modelo a partir de la entidad `Employee`, navegando a su departamento y dirección correspondiente con la notación del punto, así como la selección del atributo nombre y estado, en cada caso sobre su entidad correspondiente.



```
SELECT e
FROM Employee e
WHERE e.department.name = 'NA42' AND
e.address.state IN ('NY', 'CA')
```

Código 3: Consulta con elementos WHERE, operadores AND e IN

Sin embargo, a la hora de **proyectar** los datos, tenemos la restricción de que estos deben ser o entidades o datos persistentes, **no pueden ser colecciones**. Si queremos recuperar datos de colecciones debemos utilizar el JOIN con entidades.

Supongamos que queremos recuperar los teléfonos de todos los empleados del departamento 'NA42' y de tipo celular, en tal caso debemos realizar un JOIN entre la entidad empleado y el empleado del teléfono correspondiente (ver cláusula WHERE en Código 4 con e = p.employee).

```
SELECT p.number
FROM Employee e, Phone p
WHERE e = p.employee AND
e.department.name = 'NA42' AND
p.type = 'Cell'
```

Código 4: CROSS JOIN simple entre Employee y Phone

Al igual que con las SQL se permite el uso del operador JOIN, expresando el JOIN en función de la relación así como las variantes LEFT JOIN y JOIN FETCH (cargando adicionalmente los objetos del lado derecho del JOIN). En el ejemplo mostrado en el Código 5, podemos ver como en el uso del JOIN navegamos sobre la colección de teléfonos con empleados (e.phones).

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND
p.type = 'Cell'
```

Código 5: Sintaxis JOIN entre Employee y Phone

Inicialmente el JOIN en JPQL se puede realizar de **manera natural cuando se han definido relaciones en el modelo**. ¿Pero qué ocurre si las entidades no están relacionadas? Ya hemos visto que en JPA existe la posibilidad de crear un CROSS JOIN, con el producto cartesiano de las entidades y comparando campos (ver Código 6). Esta opción tiene peor rendimiento, es más difícil de leer y no permite su combinación con los outer JOIN, por lo que en la medida de lo posible se evitará. Frameworks como Hibernate en sus últimas versiones incluyen un inner JOIN para entidades no relacionadas, pero no es parte del estándar de JPA y no será objeto de estudio.



```

SELECT p.firstName, p.lastName, n.phoneNumber
FROM Person p, PhoneBookEntry n
WHERE
p.firstName = n.firstName AND
p.lastName = n.lastName

```

Código 6: Sintaxis CROSS JOIN entre Person y PhoneBookEntry

Dentro de las características generales, indicar que también se permite el uso de agregados, con una sintaxis muy similar a SQL, usando expresiones sobre las entidades y atributos. Supongamos que queremos recuperar los departamentos, el número de empleados de cada uno, el máximo del salario y la media de salarios de los empleados del correspondiente departamento. Para resolver esta consulta utilizamos funciones agregadas y cláusulas GROUP BY y HAVING de forma muy similar a SQL (ver Código 7).

```

SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5

```

Código 7: Consulta con agregados y cláusula GROUP BY y HAVING

Nota aclaratoria: Hibernate **no implementa correctamente el GROUP BY por entidad**, para algunos dialectos de base de datos (entre ellos Oracle). Por lo tanto en caso de error, es necesario reescribir de otra forma la consulta JPQL agrupando sobre atributos de la entidad.

Finalmente y por analogía con la definición de consultas preparadas con JDBC, se pueden parametrizar, bien con el uso del carácter interrogante e índice (e.g., ?1), o bien con variables (con la notación : delante del nombre). En el Código 8, podemos ver un par de ejemplos de consultas parametrizadas, utilizando ambas soluciones sintácticas, bien por índice o bien por nombre a la hora de definir los parámetros. Posteriormente en su ejecución, se debe dar obligatoriamente una sustitución de los mismos.

```

SELECT e
FROM Employee e
WHERE e.department = ?1 AND
e.salary > ?2

SELECT e
FROM Employee e
WHERE e.department = :dept AND
e.salary > :base

```

Código 8: Consultas parametrizadas

De manera más exhaustiva, el conjunto de funciones disponibles en JPQL son:



- Agregadas:
 - COUNT, MAX, MIN, AVG, SUM
- Cadena:
 - CONCAT, SUBSTRING, TRIM, LOWER, UPPER
 - LENGTH: longitud de array de caracteres o bytes.
 - LOCATE: índice de una cadena si está contenida en otra.
- Aritméticas:
 - ABS, SQRT, MOD,
 - INDEX: posición de un elemento en una lista ordenada
- Colecciones:
 - IS [NOT] EMPTY: si una colección está o no vacía. Ej: a.books IS EMPTY.
 - [NOT] MEMBER OF: si un elemento pertenece a una cierta colección. Ej: 'Inglés' MEMBER OF c.lenguajes
 - [NOT] IN: si un elemento pertenece (o no) a una colección. Ej: "palabra" IN :textos
 - Nota: mientras que MEMBER se utiliza para colecciones miembro de la entidad consultada, IN se puede utilizar con colecciones que son parámetros a la consulta.
 - SIZE: tamaño de una colección (específica de JPQL). Ej: SIZE(a.books) = 0
- Fecha:
 - CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESPACE (relativas a la BD)
- Expresiones CASE:
 - CASE: similar a un IF
 - Sintaxis: CASE {operand} WHEN {test_value} THEN {match_result} ELSE {miss_result} END
 - COALESCE: retorna el primer valor no nulo entre varios valores.
 - NULLIF: retorna un nulo si ambos valores son iguales.

Finalmente, debemos indicar que en JPA 2.2, dentro de JPQL, no se incluyen operadores sobre conjuntos como UNION, INTERSECT, EXCEPT o MINUS².

4.2 Subconsultas

En JPQL también se puede utilizar subconsultas en cláusulas WHERE y HAVING³, con expresiones EXISTS, ALL, ANY o SOME equivalentes a las de SQL (y sus variantes con NOT). Las subconsultas van entre paréntesis (el indentado es opcional pero se incluye por claridad) y pueden devolver un valor escalar o una colección de valores. Se describen a continuación algunos ejemplos.

En el ejemplo del Código 9, podemos ver el uso de EXISTS para devolver todo empleado cuya esposa es también un empleado

2 Algunos *frameworks*, como EclipseLink, pueden incluirlos, pero no son parte de la especificación JPA y no se usarán en la asignatura.

3 A partir de JPA 2.1 se restringe su uso a WHERE y HAVING. Su inclusión en cláusulas FROM será considerado en siguientes versiones de la especificación.



```

SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)

```

Código 9: Consulta con subconsulta y EXISTS

En el ejemplo del Código 10, se consultan aquellos empleados cuyos salarios son mayores que todos los manager de su departamento.

```

SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
    SELECT m.salary
    FROM Manager m
    WHERE m.department = emp.department)

```

Código 10: Consulta con subconsulta y ALL

Como podemos ver en ambos ejemplos, y como ocurre con las subconsultas en SQL, podemos utilizar la entidades inicial de la cláusula FROM para definir posteriormente la subconsulta, estando ambas correlacionadas.

En otros casos, puede ser necesario que la subconsulta devuelva un único valor numérico, como en el Código 11. En el ejemplo, consultamos los clientes cuya media de los precios de las órdenes de compra realizadas, es mayor de 100.

```

SELECT c
FROM Customer c
WHERE (SELECT AVG(o.price) FROM c.orders o) > 100

```

Código 11: Consulta con subconsulta con valor numérico

4.3 Definición de consultas

Para la definición de consultas se utilizan dos interfaces básicas en JPQL:

- **Query**
 - Devuelve un resultado **Object** (omitiendo el tipo concreto).



- TypedQuery

- Devuelve el resultado con tipo concreto, utilizando genericidad (interfaz nueva en JPA 2.x).

Para obtener una implementación concreta se utiliza el método de fabricación de EntityManager, teniendo claro que los resultados **NUNCA se encapsulan en ResultSet o RowSet como ocurre con JDBC.**

Las consultas que se pueden definir se catalogan en dos tipos:

1. **Dinámicas**: creadas al vuelo en tiempo de ejecución.
2. **Con nombre**: configuradas por unidad de persistencia con anotaciones o bien con XML.

Las **consultas dinámicas** cambian en tiempo de ejecución y no pueden cerrarse previamente. Es más, si no se utilizan consultas parametrizadas, sino la típica concatenación de cadenas en Java, es necesario analizarlas y generar su SQL en cada ejecución. Además esto genera problemas de rendimiento al no incluirse en la caché de SQL por parte del proveedor de persistencia y están sujetas a ataques por inyección de SQL (similar a las cuestiones ya vistas con sentencias preparadas con JDBC).

A continuación, en el Código 12, podemos ver un ejemplo de una consulta dinámica que aunque usa concatenación de cadenas, sí parametriza la consulta. El tipo returned por la consulta es Long.class y se indica en la creación de la misma, en el método createQuery.

```
@Stateless // Example JEE
public class QueryServiceBean implements QueryService {
    private static final String QUERY =
        "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND " +
        "e.name = :empName ";

    @PersistenceContext(unitName="DynamicQueries") EntityManager em;
    public long queryEmpSalary(String deptName, String empName) {
        return em.createQuery(QUERY, Long.class)
            .setParameter("deptName", deptName)
            .setParameter("empName", empName)
            .getSingleResult();
    }
}
```

Código 12: Consulta dinámica parametrizada (ejemplo JEE)

Las **consultas con nombre** utilizan la anotación @NamedQuery, y por regla general se definen en el código de la entidad con la que están mas relacionadas. Se procesan solo una vez al cargarse la clase y su nombre se asocia al contexto de persistencia, debiendo ser único en dicho contexto. En el Código 13, podemos ver un ejemplo de uso de la anotación y construcción de la consulta, que puede ser parametrizada.

```
@NamedQuery(name="Employee.findSalaryForNameAndDepartment",
    query = "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND e.name = :empName")
```

Código 13: Consulta con nombre

Si se quieren definir varias consultas se pueden agrupar utilizando la anotación @NamedQueries en la clase (ver Código 14). Como convención de nombres, se utiliza el nombre de la entidad junto con el



nombre de la operación a realizar, evitando colisiones dado que deben ser únicas en el contexto de persistencia. Por ejemplo un buscador de se denominaría <entityName>.findXXXX.

```
@NamedQueries({
    @NamedQuery(name="Employee.findAll",
        query="SELECT e FROM Employee e"),
    @NamedQuery(name="Employee.findByPrimaryKey",
        query="SELECT e FROM Employee e WHERE e.id = :id"),
    @NamedQuery(name="Employee.findByName",
        query="SELECT e FROM Employee e WHERE e.name = :name")
})
```

Código 14: Consultas múltiples con nombre

Para utilizar una consulta con nombre, previamente declarada, se utiliza el método `createNamedQuery` de la interfaz `EntityManager`. En el Código 15, podemos ver un ejemplo de uso de la consulta con nombre `Employee.findByName` con tipo retornado `Employee`.

```
public class EmployeeService {

    private EntityManager em;

    // connect an EntityManager instance...

    public Employee findEmployeeByName(String name) {
        return em.createNamedQuery("Employee.findByName",
            Employee.class)
            .setParameter("name", name)
            .getSingleResult();
    }
    // ...
}
```

Código 15: Consultas múltiples con nombre

Para las consultas parametrizadas, ya sean dinámicas o por nombre, se establecen los parámetros con métodos `setParameter` (bien indicando índice o nombre). Con los parámetros de tipo de `Time` o `Date` es necesario realizar algún ajuste adicional, indicado en un tercer argumento en la invocación.

En el Código 16, vemos un ejemplo de sustitución de parámetros con ambos casos. En el primero tenemos una consulta con nombre y sustitución por nombre, mientras que en el segundo tenemos una consulta dinámica con sustitución por índice. Dado que los parámetros son de tipo fecha, es necesario dar un tercer argumento actual de tipo `TemporalType.DATE`. **Remarcar además que los índices comienzan en 1, no en cero.**

```
return em.createNamedQuery("findEmployeesAboveSal", Employee.class)
    .setParameter("dept", dept)
    .setParameter("sal", minSal)
    .getResultList();

return em.createQuery("SELECT e " +
    "FROM Employee e " +
    "WHERE e.startDate BETWEEN ?1 AND ?2",
    Employee.class)
    .setParameter(1, start, TemporalType.DATE)
    .setParameter(2, end, TemporalType.DATE)
    .getResultList();
```

Código 16: Sustitución de parámetros por nombre e índice, tanto con consultas con nombre como dinámicas



Existe un **problema en desarrollo con las @NamedQuery**: su parseo se realiza en tiempo de ejecución al cargar el modelo. Si la consulta es errónea la generación de la fábrica de gestores de entidades se interrumpe, impidiendo desde un principio la ejecución del *framework* de persistencia. Esto es un problema bastante difícil de depurar y detectar.

Como recomendación se sugiere probar primero la consulta como texto embebido en una *query* sin nombre, y una vez validada, mover su definición a una `@NamedQuery`. Si se produce un error al modificar posteriormente una `@NamedQuery` se debe tener este problema en cuenta.

4.4 Retorno de consultas

Dependiendo del número de elementos en el resultado de una consulta podemos establecer dos tipos:

- **SingleResult**
 - Devuelve un **único** valor.
 - Se utiliza el método `getSingleResult`.
 - Si no hay resultado lanza una excepción `NoResultException` y si devuelve más de uno lanza `NonUniqueResultException` (ambas heredan de `RuntimeException`).
 - El proveedor no hace `rollback` de la transacción aunque salten estas excepciones.
- **ResultList**
 - Devuelve una lista de valores.
 - Se utiliza el método `getResultList`.
 - Si no hay resultados devuelve una lista vacía (generando menos excepciones en tiempo de ejecución que el caso previo).
 - Devuelve una lista (`List`) puesto que permite las ordenaciones.

Cuando la consulta devuelve más de una expresión (columnas) se retorna una lista con *arrays* de `Object`. Esto se utiliza con consultas sin tipo (`Query`) como podemos ver en el Código 17. Una vez retornada la lista hay que recorrer los *arrays* contenidos.

```
public void displayProjectEmployees(String projectName) {
    List result = em.createQuery(
        "SELECT e.name, e.department.name " +
        "FROM Project p JOIN p.employees e " +
        "WHERE p.name = ?1 " +
        "ORDER BY e.name") // without return class type
    .setParameter(1, projectName)
    .getResultList();

    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext(); ) {
        Object[] values = (Object[]) i.next();
        System.out.println(++count + ": " +
            values[0] + ", " + values[1]);
    }
}
```

Código 17: Consulta con varias expresiones en el resultado

Si se quiere evitar el uso de *arrays* de objetos, se pueden utilizar **expresiones de construcción**. Estas expresiones permiten generar estructuras de tipo JavaBean en tiempo de ejecución (al vuelo), de manera similar a una instanciación con el operador `new`. Es una mejor solución, al ser necesaria una declaración explícita de tipos, construyendo previamente la clase que permite la construcción.



En el Código 18, podemos ver un ejemplo de uso de expresión de construcción, generando instancias de un tipo `EmpMenu` al vuelo en la consulta. Es **muy importante**, utilizar el nombre completo cualificado a continuación de la palabra reservada NEW incluida en la sentencia JPQL, en caso contrario la clase POJO⁴ no se encontrará y se genera un error. Se omite el código correspondiente a la clase auxiliar `EmpMenu`, que deberá incluir un constructor con dos argumentos, y sus correspondiente métodos *getter* y *setter*.

```
public void displayProjectEmployees(String projectName) {
    List<EmpMenu> result =
        em.createQuery("SELECT NEW example.EmpMenu(" +
                      "e.name, e.department.name) " +
                      "FROM Project p JOIN p.employees e " +
                      "WHERE p.name = ?1 " +
                      "ORDER BY e.name",
                      EmpMenu.class)
        .setParameter(1, projectName)
        .getResultList();
    int count = 0;
    for (EmpMenu menu : result) {
        System.out.println(++count + ": " +
                           menu.getEmployeeName() + ", " +
                           menu.getDepartmentName());
    }
}
```

Código 18: Uso de expresión de construcción

Para las consultas se permite como tipo resultado: tipos primitivos, `String`, tipos JDBC, entidades, arrays de `Object` e instancias de tipos generados a través de **expresiones de construcción**.

4.5 Soporte de paginación

En muchas aplicaciones es necesario mostrar los datos paginados (un nº determinado de registros por página permitiendo el avance y retroceso en el conjunto de datos). En las consultas realizadas con JPA se da también soporte a este concepto.

Los métodos utilizados son:

- `setFirstResult()`: el primer resultado a recibir (numerado desde cero).
- `setMaxResults()`: el máximo número de resultados a retornar relativo a un punto.

Para completar la solución, es necesario mantener variables de tamaño de página y página actual. En el Código 19, se realizan consultas con `createNamedQuery`, indicando la posición del primer resultado a recibir (`currentPage * pageSize`) y el tamaño del conjunto a devolver (`pageSize`). Según se quiera avanzar de página, se incrementa el contador `currentPage`.

4 Los POJO no incluyen anotaciones JPA y no son incluidos en el fichero `persistence.xml`.



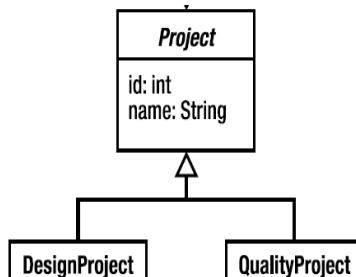
```
private long currentPage; private long maxResults; private long pageSize;  
// other code...  
  
public void init(long pageSize, String countQueryName, String reportQueryName) {  
    this.pageSize = pageSize;  
    this.reportQueryName = reportQueryName;  
    maxResults = em.createNamedQuery(countQueryName, Long.class)  
        .getSingleResult();  
    currentPage = 0;  
}  
  
public List getCurrentResults() {  
    return em.createNamedQuery(reportQueryName)  
        .setFirstResult(currentPage * pageSize)  
        .setMaxResults(pageSize)  
        .getResultList();  
}  
// method getCurrentPage(), next(), previous(), getMaxPages()
```

Código 19: Uso de expresión de construcción

Nota aclaratoria: aunque en algunos SGBD (e.g. PostgreSQL) se da soporte a esto de manera nativa a través de la sintaxis SQL para SELECT, con los parámetros LIMIT y OFFSET, en la versión 11g de Oracle no está disponible. A partir de la versión 12c de Oracle se incorporó a la sintaxis de consulta LIMIT y FETCH (ver por ejemplo <https://oracle-base.com/articles/12c/row-limiting-clause-for-top-n-queries-12cr1>) que cumplen un papel similar. Se puede consultar dicha sintaxis, pero no es aplicable en la asignatura, dada la versión de Oracle utilizada en prácticas.

4.6 Consultas con herencia

JPA permite consultas polimórficas en las jerarquías de herencia sobre entidades. Por ejemplo, si tomamos la jerarquía de herencia del Dibujo 1, tenemos una especialización de proyectos en dos subtipos.



Dibujo 1: Jeraquía de herencia de proyectos. Figura extraída de [Keith & Schincariol, 2009]

Si realizamos una consulta de entidades proyecto (`Project`), obtendremos instancias de ambos conjuntos (`DesignProject` o `QualityProject`). Si se quiere restringir el resultado de la consulta a un solo tipo basta con especificarlo en la cláusula FROM. En el ejemplo del Código 20 se muestra cómo obtenemos todos los proyectos, indicando en la cláusula FROM la entidad a consultar.



```
SELECT p  
FROM Project p  
WHERE p.employees IS NOT EMPTY
```

Código 20: Consulta polimórfica

Si por otro lado, queremos restringir a varios (o algún tipo concreto), es necesario usar la función `TYPE(expr.de entidad)` como se muestra en Código 21.

```
SELECT p  
FROM Project p  
WHERE TYPE(p) = DesignProject OR TYPE(p) = QualityProject
```

Código 21: Consulta polimórfica con expresión de tipo

También es posible realizar un “downcast” en la consulta, cuando estamos consultando sobre la superclase pero necesitamos acceder a alguna propiedad particular de la subclase, utilizando la función `TREAT`. Por ejemplo, en la siguiente consulta del Código 22:

```
SELECT p  
FROM Project p  
WHERE TREAT(p AS QualityProject).qaRating > 4  
OR TYPE(p) = DesignProject
```

Código 22: Consulta con downcast utilizando `TREAT`

Finalmente incluso puede usarse el `TREAT` en el propio `JOIN`, para limitar el conjunto de entidades a un subtipo (ver Código 23). En este ejemplo, de los empleados se hace el *join* con sus proyectos pero de tipo calidad, permitiendo su acceso a través de la variable `q`.

```
SELECT e, q.name, q.qaRating  
FROM Employee e JOIN TREAT(e.projects AS QualityProject) q  
WHERE q.qaRating > 4
```

Código 23: Consulta con `TREAT` en el `JOIN`

4.7 Bulk update y delete

Las operaciones de actualización y borrado se pueden ejecutar directamente contra la base de datos, a través de operaciones “a granel”. Para ello se definen como consultas, con `createQuery`, y se ejecutan a través del método `executeUpdate`.

En el Código 24 y Código 25, se muestran dos ejemplos de operaciones de actualización y borrado respectivamente.



```
public void assignManager(Department dept, Employee manager) {  
    em.createQuery("UPDATE Employee e " +  
        "SET e.manager = ?1 " +  
        "WHERE e.department = ?2")  
        .setParameter(1, manager)  
        .setParameter(2, dept)  
        .executeUpdate();  
}
```

Código 24: Ejemplo de bulk update

```
public void removeEmptyProjects() {  
    em.createQuery("DELETE FROM Project p " +  
        "WHERE p.employees IS EMPTY")  
        .executeUpdate();  
}
```

Código 25: Ejemplo de bulk delete

Se debe tener en cuenta que ambas realizan las operaciones **directamente** contra la base de datos, no actualizando las instancias de las entidades persistentes en memoria, por lo que **deben utilizarse con mucho cuidado**.

DELETE en JPA borra un conjunto de entidades (`remove` en JPA eliminaba solo una entidad / tupla), pero no se aplica `cascade` a las entidades relacionadas. Si se violan ciertas restricciones, se lanza una excepción `PersistenceException`. Por lo tanto puede ser necesario aplicar previamente actualizaciones para resolver el conflicto, como se muestra en el Código 26.

```
DELETE FROM Department d  
WHERE d.name IN ('CA13', 'CA19', 'NY30')  
  
-- error with foreign key integrity constraint  
  
-- we need to update before...  
UPDATE Employee e  
SET e.department = null  
WHERE e.department.name IN ('CA13', 'CA19', 'NY30')  
-- and now we can run the delete again...
```

Código 26: Error en bulk delete y resolución con bulk update previo

Nota avanzada: en relaciones varios a varios, con una tabla intermedia (*join table*) gestionada por el framework (no hay entidad asociada a esa tabla), sí que se generan sentencias adicionales que actualizan o borran las filas correspondientes en dicha tabla. Por ejemplo, supongamos una relación varios a varios entre una tabla A y B, teniendo en la base de datos una tabla intermedia A_B con las claves foráneas correspondientes a dichas tablas formando su clave primaria. Si realizamos un DELETE BULK sobre entidades A o B, se genera y ejecuta previamente una operación SQL adicional DELETE sobre dicha tabla intermedia, para garantizar la integridad referencial (en caso contrario se generaría un error en la base de datos).



4.8 Consultas nativas

Finalmente, podemos encontrarnos limitados en el uso de JPQL respecto a las consultas SQL que podemos ejecutar. Si queremos realizar **consultas nativas, específicas del SGBD** que estemos utilizando en nuestra aplicación, y evitar la traducción al dialecto correspondiente por parte del proveedor de persistencia, utilizaremos consultas nativas.

En estas consultas se devuelve siempre una `Query` sin tipo. Pero por otro lado, sí se pueden definir con nombre las consultas, utilizando la anotación `@NamedNativeQuery`, y usar `createNamedQuery` para obtener un resultado tipado (`TypedQuery`).

Estas consultas pueden devolver entidades gestionadas, pero es responsabilidad nuestra recuperar todos los datos necesarios. En este caso se puede pasar como argumento adicional el tipo entidad a recuperar.

En el Código 27, se presenta una consulta nativa con nombre (e.g. consulta jerárquica particular de Oracle, con sintaxis particular). En este ejemplo se recuperan entidades de tipo `Employee`. Mientras que en el Código 28, se presenta su uso en código.

```
@NamedNativeQuery(  
    name="OrgStructureBean.orgStructureReportingTo",  
    query="SELECT emp_id, name, salary, manager_id, dept_id, address_id " +  
        "FROM emp " +  
        "START WITH manager_id = ? " +  
        "CONNECT BY PRIOR emp_id = manager_id",  
    resultClass = Employee.class  
)
```

Código 27: Ejemplo de definición de consulta nativa con nombre

```
public class OrgStructureBean implements OrgStructure {  
  
    EntityManager em;  
  
    public List<Employee> findEmployeesReportingTo(int managerId) {  
        return em.createNamedQuery("OrgStructureBean.orgStructureReportingTo",  
            Employee.class)  
            .setParameter(1, managerId)  
            .getResultList();  
    }  
}
```

Código 28: Ejemplo de uso de consulta nativa con nombre

Por otro lado, se puede ejecutar también operaciones INSERT, UPDATE y DELETE de forma nativa de forma similar (ver Código 29). Aún siendo muy optimas, no actualizan las entidades, ni la caché, y tampoco se pueden combinar con interceptores del ciclo de vida de las entidades.



```
public class LoggerBean implements Logger {  
    private static final String INSERT_SQL =  
        "INSERT INTO message_log (id, message, log_dttm) " +  
        "VALUES(id_seq.nextval, ?, SYSDATE)";  
  
    private static final String DELETE_SQL =  
        "DELETE FROM message_log";  
  
    private EntityManager em;  
  
    public void logMessage(String message) {  
        em.createNativeQuery(INSERT_SQL)  
            .setParameter(1, message)  
            .executeUpdate();  
    }  
  
    public void clearMessageLog() {  
        em.createNativeQuery(DELETE_SQL)  
            .executeUpdate();  
    }  
}
```

Código 29: Ejemplo de uso de operaciones *INSERT* y *DELETE* nativas

Una diferencia adicional con el uso de operaciones nativas, es que se permite el uso de parámetros pero solo por posición, no por nombre.

Nota: Hibernate permite parámetros por nombre pero no es parte de la especificación JPA.

Si el mapeo del resultado de una consulta no encaja con ninguna entidad, sino con un POJO, se puede indicar el mapeo a través de la anotación `@SqlResultSetMapping`.

Por ejemplo, para recoger el resultado de una consulta en un POJO de tipo `AuthorValue`, se define (ver Código 30) en la entidad relacionada (e.g., `Author`):

```
@SqlResultSetMapping(  
    name = "AuthorValueMapping",  
    classes = @ConstructorResult(  
        targetClass = AuthorValue.class,  
        columns = {  
            @ColumnResult(name = "id", type = Long.class),  
            @ColumnResult(name = "firstname"),  
            @ColumnResult(name = "lastname"),  
            @ColumnResult(name = "numBooks", type = Long.class)}))
```

Código 30: Ejemplo de uso de mapeo POJO con consulta nativa

Cuando se ejecuta la consulta (ver Error: no se encontró el origen de la referencia), en lugar de pasar como argumento un tipo, se pasa el nombre del mapeo, en el ejemplo “`AuthorValueMapping`”. De esta forma se busca un constructor en la clase POJO que coincida con las columnas resultado de la consulta SQL nativa y definidas en la anotación (en el ejemplo, un constructor con cuatro argumentos, correspondencia uno a uno).

Este mecanismo tiene ciertas similitudes con el uso de expresiones de construcción, ya visto previamente.



5. Grafos de entidades

5.1 Planteamiento del problema

Un problema con JPA es el número de consultas generadas cuando estamos con una estrategia de carga SELECT (consulta adicional por atributo/entidad/relación). Esto genera el problema denominado "N+1 consultas", en lugar de resolverlo en una única consulta con un JOIN. Esto se multiplica con el número de clientes en paralelo que podemos tener en ejecución.

Recordando que por defecto se establece como valores de carga para las relaciones:

- @OneToOne / @ManyToOne → EAGER.
- @OneToMany / @ManyToMany → LAZY.

Esta propiedad puede ser redefinida desde un principio como LAZY o EAGER sobre el modelo. Sin embargo esta **definición estática** también puede ser problemática, puesto que se arrastra posteriormente a la hora de realizar todas las operaciones (de hecho se considera como grafo "por defecto" a cargar). Y puede no ser la solución ideal en todos los casos.

Aunque esto **se puede resolver parcialmente con el uso particular para ciertas operaciones de JOIN FETCH con JPQL o Criteria API**, forzando la recuperación de datos, y en cierto sentido redefiniendo la estrategia de carga por defecto.

Pero esta solución multiplica el número de consultas y métodos particulares definidos por parte del programador. Este es un problema que ha existido siempre en JPA desde versiones previas.

Sin embargo desde la versión 2.1, se introduce el concepto de los grafos de entidades como solución alternativa.

5.2 Definición y uso de grafos de entidades

Los grafos de entidades surgen entonces como **planes de recuperación o grupos de campos persistentes que se recuperan al mismo tiempo**. Redefinen en **tiempo de ejecución** el `fetch` sobre los atributos. En cierta forma, se puede entender que realizan adicionalmente un conjunto de *joins* de manera transparente en las relaciones, cumpliendo el papel de plantilla a utilizar en las consultas o búsquedas de las entidades (en cierto sentido utilizando una estrategia de carga JOIN para la recuperación de datos en relaciones).

La definición de un grafo de entidades se puede realizar declarativa o programáticamente, a través de anotaciones o bien con código, siendo independiente de la consulta donde se utiliza posteriormente.

Se pueden utilizar con el método `EntityManager.find` o como parte de una consulta JPQL (o Criteria API), especificando el grafo de entidades y una pista de cómo realizar la carga.

Se especifica cómo se quiere realizar la carga con dos propiedades:

- `javax.persistence.fetchgraph`: todos los atributos indicados en el grafo de entidades como EAGER (utilizando `join`, si se requiere). Los atributos no especificados se tratan como LAZY.
- `javax.persistence.loadgraph`: todos los atributos indicados en el grafo de entidades como EAGER (utilizando `join`, si se requiere). Los atributos no especificados se tratan según su valor por defecto o su valor especificado explícitamente en el modelo de clases, si existe.

Nota aclaratoria: los atributos claves primarias (simples o compuestas) y los atributos versión (para bloqueo optimista) son siempre recuperadas y no es necesario incluirlas (se puede hacer pero es redundante). Para una información más detallada de la semántica de carga, se recomienda una lectura de la especificación JPA 2.1, secciones **3.7.4.1 – Fetch Graph Semantics** y **3.7.4.2 – Load Graph**



Semantics. Adicionalmente, señalar que **frameworks como Hibernate, cargan por defecto todos los atributos básicos (@Basic) de forma ansiosa (EAGER)** aunque no se indiquen en la lista de atributos.

La definición del grafo se realiza declarativamente en la clase correspondiente (de manera similar a una @NamedQuery), como se muestra en el Código 31 a través de un @NamedEntityGraph:

```
@NamedEntityGraph(
    name = "peliculasConActores", // nombre del grafo
    attributeNodes = { // lista de atributos a cargar obligatoriamente en el grafo
        @NamedAttributeNode("actores")
        // suponemos un atributo @OneToMany con dicho nombre "actores" que mantiene
        // la relación de una película con sus actores
    }
)
```

Código 31: Ejemplo simple de definición de grafo de entidades

Por otro lado se permite también definir carga de relaciones en segundo nivel, a través del concepto de subgrafos como se muestra en el Código 32. En este ejemplo junto con las películas y actores, se carga el subgrafo correspondientes a los premios de los actores en dichas películas. Para ello se utiliza la sintaxis adicional de la anotación @NamedSubgraph.

```
@NamedEntityGraph(
    name = "peliculasConActoresYPremios", // nombre del grafo
    attributeNodes = {
        @NamedAttributeNode(value = "actores", subgraph = "actoresPelicula")
        // suponemos un atributo @OneToMany con dicho nombre "actores" que mantiene
        // la relación de una película con sus actores
        // se vincula con un subgrafo que permite navegar a los premios de los actores
    },
    subgraphs = {
        @NamedSubgraph(
            name = "actoresPelicula", // nombre del subgrafo con los actores
            attributeNodes = {
                @NamedAttributeNode("premios")
                // suponemos un atributo @OneToMany con nombre "premios" que mantiene
                // la relación de premios de esos actores
            }
        )
    }
)
```

Código 32: Ejemplo de definición de grafo y subgrafos de entidades

Como atajo, si se quieren cargar todos los atributos de una entidad, en lugar de enumerar uno a uno con @NamedAttributeNode, se puede utilizar @NamedEntityGraph(includeAllAttributes=true). Este atributo tiene efectos solo sobre los atributos de la entidad anotada (raíz) y no sobre los subgrafos.

Incluso si la entidad solo contiene **atributos básicos (sin relaciones)**, que tienen carga por defecto EAGER, se puede declarar @NamedEntityGraph sin más.

Se pueden definir varios grafos en una misma clase (anotación @NamedEntityGraph) añadiéndolas dentro de una anotación de clase javax.persistence.NamedEntityGraphs, de manera similar a la definición múltiple de consultas con nombre.

El uso de los grafos se lleva a cabo finalmente en búsquedas con find o consultas JPQL (o Criteria API), a través de código. En el Código 33, podemos ver un ejemplo de uso de grafo, a través del método de



búsqueda `find` por clave primaria. En el siguiente ejemplo (ver Código 34) se define un método de consulta que permite parametrizar tanto el nombre del grafo a utilizar (e.g. podríamos usar los nombres de alguno de los dos grafos declarados previamente), como la pista de carga a aplicar. Se supone que se utiliza una consulta con nombre con JPQL (e.g. "`Pelicula.findAll`"), previamente declarada también en la clase.

```
// previamente se ha establecido el idPelicula
EntityGraph graph = em.getEntityGraph("peliculasConActoresYPremio");
Map<String, EntityGraph> hints = new HashMap<>(); // mapa de propiedades "con pistas"
hints.put("javax.persistence.fetchgraph", graph); // establecer carga y grafo
return em.find(Pelicula.class, idPelicula, hints);
```

Código 33: Ejemplos de búsqueda con `find` usando grafo de entidades y pista

El valor de pista será: `javax.persistence.fetchgraph` o `javax.persistence.loadgraph`.

Su uso con Criteria API es muy similar, parametrizando igualmente el objeto `query` con el método `setHint`.

```
public List<Pelicula> consultarPelículas(String nombreGrafo, String pista) {
    return entityManager.createNamedQuery("Pelicula.findAll")
        .setHint(pista, entityManager.getEntityGraph(nombreGrafo))
        .getResultList();
}
```

Código 34: Ejemplos de consulta parametrizada con grafo de entidades y pista

La diferencia de utilizar un grafo u otro, nos llevaría a que los datos cargados en memoria son distintos, y dependiendo del posterior acceso a las relaciones, ya no fuese necesario (o sí) realizar consultas SELECT adicionales para cargar datos.

Dado que la definición declarativa puede ser pesada de realizar (con un uso abusivo de anotaciones), se permite también la definición programada, utilizando las metaclasses del modelo. Dichas metaclasses se generan automáticamente por el plugin Dali de Eclipse, y se identifican por tener el mismo nombre que la clase persistente seguida de un guión bajo (ver Apartado 7.1. Metaclasses para una descripción más amplia sobre las metaclasses). En nuestro ejemplo, suponiendo que tenemos la siguiente metaclass `Pelicula_` en el Código 35:

```
@StaticMetamodel(Pelicula.class)
public abstract class Pelicula_ {
    public static volatile SingularAttribute<Pelicula, Integer> id;
    public static volatile SetAttribute<Pelicula, Premio> premios;
    public static volatile SingularAttribute<Pelicula, String> nombre;
    public static volatile SetAttribute<Pelicula, Actor> actores;
    public static volatile SetAttribute<Pelicula, MDirector> directores;
}
```

Código 35: Ejemplo de metaclass `Pelicula_`

Podríamos definir un grafo incluyendo los subgrafos adicionales a las relaciones `@OneToMany` que tenemos a premios, actores y directores) de la forma mostrada en el Código 36, reutilizando la definición previa de metaclass:



```
EntityGraph<Movie> fetchAll = entityManager.createEntityGraph(Pelicula.class);
fetchAll.addSubgraph(Pelicula_.actores);
fetchAll.addSubgraph(Pelicula_.directores);
fetchAll.addSubgraph(Pelicula_.premios);
```

Código 36: Ejemplo de grafo de entidades programado

Posteriormente podríamos utilizar el grafo generado de igual forma que si se hubiese definido declarativamente con anotaciones.

En el uso de los grafos de entidades (y de los JOIN FETCH) con distintos frameworks, como Hibernate, se puede obtener un error del tipo `org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple Bags`. Este error se produce cuando se hacen varios JOIN FETCH, y se corrige utilizando `Set` en lugar de `List`, a la hora de indicar el tipo en relaciones a varios.

6. Auditoría y optimización

Un problema general al trabajar con consultas (y con el resto de operaciones vinculadas a SQL desde JPA) es cómo **obtener el conjunto de estadísticas** en desarrollo para poder **optimizar** nuestras aplicaciones. Podemos además combinar los mecanismos de log, ya utilizados en la asignatura habilitando la generación y visualización de los mismos, **incluyendo además el SQL generado internamente por nuestro framework**.

En concreto, **desde Hibernate**, se puede resolver configurando adecuadamente en el fichero `persistence.xml`, tanto los valores para la generación del SQL, como la generación de estadísticas. Son propiedades booleanas (`true` o `false`) que pueden activarse o desactivarse (ver Código 37).

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

Código 37: Ejemplo de configuración de propiedades de auditoria con Hibernate

De esta forma, en posteriores ejecuciones, **se mostrará en el log generado** (dependiendo de cómo hemos configurado el fichero de propiedades de SLF4J y Log4J) tanto las operaciones SQL que se envían a la base de datos como el resumen estadístico de consultas generadas. El acceso a las estadísticas también se permite en Hibernate, a través de su API, pero no se estudiará por quedar fuera del ámbito de JPA.

El parámetro “`hibernate.use_sql-comments`” cumple la finalidad de añadir un texto adicional que permite **identificar las consultas en el log**, diferenciando unas de otras, cuando el SQL generado no lo permite. Para personalizar dicho texto, sobre la consulta ejecutar se puede invocar al método `setHint` pasando un texto a mostrar (Ej: `query.setHint("org.hibernate.comment", "Este es mi comentario asociado a la consulta X");`).

Es **MUY IMPORTANTE comprobar que efectivamente el framework genera las SQL esperadas, tanto en forma como en número**. En muchos casos el número puede ser mayor del esperado por un mal uso del *framework*, siendo **necesarias labores de revisión y optimización posteriores**.



Por ejemplo (ver Código 38): supongamos que tenemos la entidad Autor que tiene muchos libros. Si consultamos TODOS los autores (supongamos 11 autores en la base de datos) con una única consulta, y en un bucle accedemos al número de libros de cada autor, podríamos pensar que con la consulta inicial de autores ya es suficiente. Sin embargo, y dado que tenemos una relación uno a varios con libros (con carga por defecto perezosa), cada vez que accedemos a los libros de cada autor, se genera una consulta adicional.

```
List<Autor> autores = // 1 consulta ej: SELECT a FROM Autor a ... pero sin los libros
for (Autor autor : autores) {
    System.out.println(autor.getName() + " tiene " + autor.getBooks().size());
    // 1 consulta SQL adicional para traer los libros de cada autor (11) y contarlos
}
```

Código 38: Ejemplo de código que genera el problema de n+1

Para detectar este problema:

- Si hemos activado la visualización del SQL generado en el log, se mostrarán las **12 consultas SELECT generadas**. Una para traer todos los autores y 11 para traer los libros de cada autor. Si usamos las pistas con el comentario de texto asociado será más fácil identificar las consultas.
- Si hemos activado las estadísticas se observará en la salida las siguientes líneas (obviamente con tiempos de ejecución diferentes):


```
1091216 nanoseconds spent preparing 12 JDBC statements;
11118842 nanoseconds spent executing 12 JDBC statements;
```

 - Aquí se puede observar que ha generado **12 sentencias JDBC** frente a la idea optimista de utilizar solo **1 consulta**.

Queda como labor posterior resolver el problema mediante las técnicas ya vistas previamente (preferiblemente JOIN FETCH o uso de grafo de entidades en la consulta inicial de todos los autores).

Concluyendo, es responsabilidad del desarrollador auditar y revisar dicha información con el fin de ajustar y optimizar adecuadamente el código puesto en explotación.



7. Anexo 1. Criteria API

Mientras que **JPQL sigue un modelo más tradicional** de utilizar un “dialecto” de SQL para la realización de consultas (finalmente, sobre un modelo de objetos), Criteria API da una solución **muy diferente**.

Esta API ofrece la posibilidad de construir consultas a través de **objetos Java** en lugar de **cadenas de texto**, como se realiza con JPQL. Dichos objetos están en consonancia con un modelo de clases, que refleja el modelado de la construcción de consultas de una forma abstracta.

En el Código 39, se muestra la solución más tradicional con JPQL mientras que en el Código 40, se muestra su equivalente con Criteria API, instanciando los objetos necesarios que modelan consultas.

```
SELECT e
FROM Employee e
WHERE e.name = 'John Smith'
```

Código 39: Ejemplo simple de JPQL

Con Criteria API, se instancian objetos correspondientes a las distintas partes de una consulta SQL: la cláusula FROM se corresponde con la instancia de `Root`, la cláusula SELECT se corresponde con el método `select` y la cláusula WHERE se corresponde con el método `where` (ver Código 40).

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
cq.select(emp)
    .where(cb.equal(emp.get("name"), "John Smith"));
```

Código 40: Ejemplo simple de Criteria API

Debemos señalar que la solución con Criteria API suele **exigir escribir mucho más código Java** (la solución es “*mucho más orientada objetos*” pero más “*verbose*”). El conjunto de clases que incluye la API para el modelado de las consultas es bastante complejo como se presenta en la Ilustración 1⁷.

7 Ver <http://docs.oracle.com/javaee/7/tutorial/persistence-criteria.htm#GJITV>



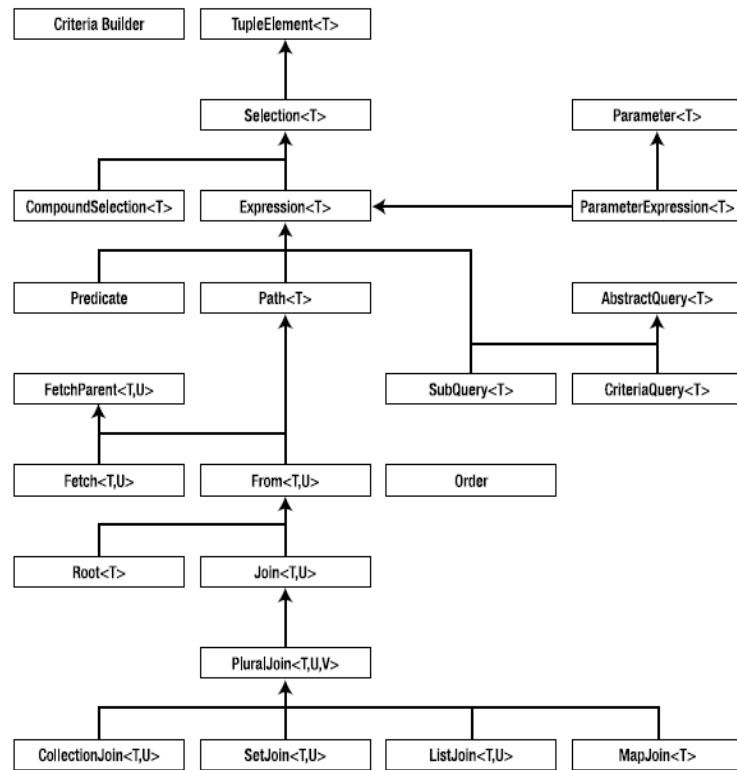


Ilustración 1: Diagrama de clases básicas de Criteria API

7.1 Metaclasses

Para poder realizar las consultas es necesario generar las metaclasses correspondientes a la definición de metadatos de las entidades persistentes utilizadas en las consultas.

La convención de nombres es que se llaman igual que la entidad con un guión bajo detrás. Estas metaclasses son accesibles por parte del gestor de entidades y necesarias para las operaciones JOIN. Para los atributos simples se utiliza la clase `SingleAttribute` mientras que para las relaciones se utiliza `SetAttribute`. El uso de genericidad es intensivo en la declaración de las metaclasses.

A continuación se muestra un ejemplo de definición de metaclass `Pet_` (ver Código 41) para la entidad `Pet` definida en (ver Código 42).

```

@StaticMetamodel(Pet.class)
public class Pet_ {
    public static volatile SingularAttribute<Pet, Long> id;
    public static volatile SingularAttribute<Pet, String> name;
    public static volatile SingularAttribute<Pet, String> color;
    public static volatile SetAttribute<Pet, Owner> owners;
}

```

Código 41: Ejemplo de metaclass `Pet`



```
@Entity
public class Pet {
    @Id
    protected Long id;
    protected String name;
    protected String color;
    @OneToMany
    protected Set<Owner> owners;
    //...
}
```

Código 42: Ejemplo de clase `Pet` de la que se genera una metaclasé `_Pet`

Para la generación de estas clases, se puede delegar en el plugin *Dali*⁸ para Eclipse que permite la generación automática de estas metaclasses. Para ello, en las propiedades del proyecto, en la sección *JPA*, se debe indicar en el diálogo, en la opción *Canonical metamodel*, el directorio destino para la generación de las metaclasses.

7.2 Construcción de consultas

Se dispone de dos clases para construcción de las consultas:

- `CriteriaBuilder`: se utiliza para construir consultas, selecciones, expresiones, predicados y ordenaciones.
 - Ej: `EntityManager em = ...;`
`CriteriaBuilder cb = em.getCriteriaBuilder();`
- `CriteriaQuery`: permite la definición de la consulta con tipo.
`Ej: CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);`

Para obtener la raíz de los datos que dirigen la consulta (cláusula `FROM`), se utiliza el objeto `CriteriaQuery` obtenido a partir de `CriteriaBuilder`, y el método `from`, pasando como argumento la entidad raíz sobre la que se proyecta la consulta. En el Código 43, se muestra un ejemplo concreto de uso.

```
EntityManager em = ....;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

Código 43: Construcción de la cláusula `FROM` con `Criteria API`

Si la consulta se proyecta sobre el `JOIN` de dos entidades (o tablas) se utilizan las metaclasses anteriormente descritas y el método `join` sobre la raíz de la consulta. En el ejemplo del Código 44, se realizan el `join` entre las entidades `Pet` y `Owner` con relación `@OneToMany`.

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

Código 44: Construcción de la cláusula `FROM` con `JOIN` utilizando metaclasses

8 Si se han seguido los guiones de prácticas relativos a la instalación de plugins, debería estar disponible actualmente en Eclipse.



Si se quieren encadenar varios JOIN, no es necesario instanciar un objeto Join, para cada operación, sino que se pueden encadenar como se muestra en Código 45.

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);  
  
Root<Pet> pet = cq.from(Pet.class);  
//Chaining join  
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.addresses);
```

Código 45: Construcción de la cláusula FROM con JOIN encadenados

Para la construcción de la SELECT se utilizan objetos de tipo Path combinando el uso de métodos get con argumento el atributo de la metaclasa: bien con valor simple (SingularAttribute) o bien con colección (CollectionAttribute, SetAttribute, ListAttribute o MapAttribute).

En el Código 46, se muestra un ejemplo de construcción de path para una consulta. En este ejemplo se realiza la proyección sobre la columna name de la tabla PET.

```
CriteriaQuery<String> cq = cb.createQuery(String.class);  
Metamodel m = em.getMetamodel();  
  
EntityType<Pet> Pet_ = m.entity(Pet.class);  
Root<Pet> pet = cq.from(Pet.class);  
  
cq.select(pet.get(Pet_.getSingularAttribute("name", String.class)));
```

Código 46: Construcción de path para construcción de SELECT

Para construir la cláusula WHERE, es necesario evaluar expresiones que restringen los resultados. Las expresiones se utilizan no solo en estas cláusulas, sino también en SELECT y HAVING. Para la construcción de expresiones se utilizan las interfaces Expression y CriteriaBuilder.

7.3 Expresiones

El conjunto de expresiones básico es el que se presenta en la Tabla 1:



Conditional Method	Description
equal	Tests whether two expressions are equal
notEqual	Tests whether two expressions are not equal
gt	Tests whether the first numeric expression is greater than the second numeric expression
ge	Tests whether the first numeric expression is greater than or equal to the second numeric expression
lt	Tests whether the first numeric expression is less than the second numeric expression
le	Tests whether the first numeric expression is less than or equal to the second numeric expression
between	Tests whether the first expression is between the second and third expression in value
like	Tests whether the expression matches a given pattern

Tabla 1: Expresiones en Criteria API

En el siguiente ejemplo (ver Código 47), se presenta ejemplos del uso de expresiones IN y BETWEEN.

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
// using expression
cq.where(pet.get(Pet_.color).in("brown", "black"));

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
// using expression
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

Código 47: Uso de expresiones IN y BETWEEN

Si se quieren combinar varias expresiones podemos definir predicados compuestos utilizando los típicos operadores AND, OR y NOT como métodos (ver Código 48):

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);

cq.where(cb.equal(pet.get(Pet_.name), "Fido").and(cb.equal(pet.get(Pet_.color), "brown")));
```

Código 48: Expresiones combinadas con operador AND

También se pueden ordenar los datos resultantes (ORDER BY) (ver Código 49), así como aplicar agrupación y filtrado sobre las agrupaciones (GROUP BY y HAVING) (ver Código 50).



```

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = cq.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName)),
           owner.get(Owner_.firstName)));

```

Código 49: Uso de ORDER BY

```

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);

cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).value("brown").value("blonde")));

```

Código 50: Uso de GROUP BY y HAVING

7.4 Ejecución de consultas

En la ejecución de las consultas con Criteria API se siguen los siguientes pasos:

1. Se obtiene el `CriteriaQuery` a partir de un `CriteriaBuilder` con su método `createQuery`.
2. Se prepara la consulta creando un objeto `TypedQuery<T>` donde `T` representa el tipo resultado de la consulta.
3. Se ejecuta la consulta con los métodos `getSingleResult` o `getResultList` sobre el objeto `TypedQuery<T>`.

En el Código 51, se muestra un par de ejemplos.

```

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
//...define...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
//...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();

```

Código 51: Ejemplos de ejecución de consultas con resultado simple o lista

7.5 Subconsultas

Al igual que con JPQL se pueden definir subconsultas. En este caso, partiendo del método `subquery`, disponible en `CriteriaQuery` por herencia (de la interfaz `AbstractQuery`) se puede definir la subconsulta a partir de la consulta previa.

En el siguiente ejemplo, en el Código 52, se muestra una definición de subconsulta no correlacionada.



```
// crea criteria query con la raíz Customer
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> goodCustomer = q.from(Customer.class);

// crea una subconsulta con la raíz Customer
// la subconsulta es tipada en función de su tipo de retorno
Subquery<Double> sq = q.subquery(Double.class);
Root<Customer> customer = sq.from(Customer.class);

// el resultado de la primera consulta depende de la subconsulta
q.where(cb.lt(
    goodCustomer.get(Customer_.balanceOwed),
    sq.select(cb.avg(customer.get(Customer_.balanceOwed))))));
q.select(goodCustomer);

/* Equivalente a...
SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (
SELECT AVG(c.balanceOwed) FROM Customer c) */
```

Código 52: Ejemplos de subconsulta (no correlacionada) con Criteria API

También se pueden resolver subconsultas correlacionadas como en el Código 53.

```
// crea criteria query con la raíz Employee
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);

// crea subconsulta con la raíz Employee
Subquery<Employee> sq = q.subquery(Employee.class);
Root<Employee> spouseEmp = sq.from(Employee.class);

// la subconsulta referencia a la raíz de la consulta contenedora
sq.where(cb.equal(spouseEmp, emp.get(Employee_.spouse)))
.select(spouseEmp);

// una condición exists es aplicada al resultado de la subconsulta:
q.where(cb.exists(sq));
q.select(emp).distinct(true);

/* Equivalente a...
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
SELECT spouseEmp
FROM Employee spouseEmp
WHERE spouseEmp = emp.spouse) */
```

Código 53: Ejemplos de subconsulta (correlacionada) con Criteria API



7.6 Comparativa JPQL vs. Criteria API

Concluyendo, podemos realizar una breve comparativa entre las dos soluciones presentadas. Mientras que JPQL es más fácil de usar, debido a su similitud con SQL, nos encontramos con la falta de chequeos estáticos sobre el código, **producíendose los errores en tiempo de ejecución**.

Sin embargo Criteria API, permite una definición más dinámica de las consultas y son seguras al **comprobarse en compilación**. El uso de las metaclasses también facilita la refactorización del código. Pero esta solución exige escribir **mucho más código** y es necesario un esfuerzo adicional con la creación de las metaclasses. Esta última pega se elimina si se utilizan herramientas que las generan dinámicamente.

En la siguiente Tabla 2, se muestra la equivalencia entre operaciones JPQL y las clases/métodos correspondientes en Criteria API.

JPQL Clause	Criteria API Interface	Method
SELECT	CriteriaQuery	select()
	Subquery	select()
FROM	AbstractQuery	from()
WHERE	AbstractQuery	where()
ORDER BY	CriteriaQuery	orderBy()
GROUP BY	AbstractQuery	groupBy()
HAVING	AbstractQuery	having()

Tabla 2: Equivalencia entre JPQL y Criteria API

8. Anexo 2. Usos avanzados⁹

8.1 Uso de procedimientos almacenados desde JPA

Aunque hemos visto diversos mecanismos de consulta y acceso directo a los datos, el tema del rendimiento sigue siendo un punto flaco en JPA, en particular cuando no se conoce o no se sabe exactamente cómo trabaja la especificación y el *framework*.

Una cuestión a tomar en cuenta, es que en muchos casos, sigue siendo mucho más rápido realizar procesamiento **en el lado del servidor de forma nativa**, que en el cliente. El implementar excesiva lógica de negocio desde nuestra aplicación cliente en Java, no siempre es positivo, en particular desde el aspecto del rendimiento. Como hemos visto en otros temas, la solución de implementar procedimientos almacenados y *triggers* ofrece una posibilidades muy potentes que sería conveniente reutilizar.

⁹ Los anexos de estas secciones contienen material adicional que **NO entra como materia de evaluación**, pero se deja disponible para su consulta posterior, por si fuera de utilidad para los alumnos.



Mientras que en JPA 2.0, estábamos limitados al uso de consultas nativas (y a su uso para la invocación de procedimientos almacenados), en particular a partir de JPA 2.1 podemos invocar directamente a procedimientos almacenados desde nuestro código Java.

Para ello se introduce un concepto similar a las consultas con nombres, a través de la anotación `@NamedStoredProcedureQuery` y de la clase `StoredProcedureQuery`.

Mediante una consulta con nombre, a procedimiento almacenado, se puede establecer declarativamente, el enlace de nuestra aplicación con un procedimiento almacenado definido en nuestro SGBD. Esta solución está inspirada en el uso ya habitual de consultas con nombre.

En el siguiente ejemplo (ver Código 54) se define la interfaz del procedimiento almacenado a invocar. Se da un nombre, y se indica el nombre del procedimiento almacenado, parámetros y valor de retorno (además de IN y OUT, se definen otros modos como INOUT o REF_CURSOR).

```
@NamedStoredProcedureQuery(
    name = "calcular",
    procedureName = "calcular",
    parameters = {
        @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "x"),
        @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "y"),
        @StoredProcedureParameter(mode = ParameterMode.OUT, type = Double.class, name = "sum") })
```

Código 54: Ejemplo de definición de un `@NamedStoredProcedureQuery`

Para la invocación del procedimiento, se realiza desde código, utilizando el `EntityManager` y su método `createNamedStoredProcedureQuery`, que devuelve un objeto `StoredProcedureQuery`. Se muestra un ejemplo de invocación en el Código 55:

```
StoredProcedureQuery query = em.createNamedStoredProcedureQuery("calcular");
query.setParameter("x", 1.23d); // establece parámetro
query.setParameter("y", 4.56d); // establece parámetro
query.execute(); // ejecuta el procedimiento
Double sum = (Double) query.getOutputParameterValue("sum"); // recoge resultado
```

Código 55: Ejemplo de invocación de un procedimiento almacenado

La invocación a procedimiento almacenado también se puede hacer directamente desde código, sin utilizar una declaración previa. En el siguiente código (ver Código 56) se muestra un ejemplo utilizando una invocación a función que devuelve un cursor con el que rellenar la lista resultado.

```
// Establece el procedimiento (función en el ejemplo) y el tipo de retorno
StoredProcedureQuery query = em.createStoredProcedureQuery("get_books", Book.class);
// Establece el uso de REF_CURSOR como valor de retorno con el que llenar la lista de libros
query.registerStoredProcedureParameter(1, void.class, ParameterMode.REF_CURSOR);
query.execute();
List<Book> books = (List<Book>) query.getResultList();
```

Código 56: Ejemplo de invocación programada de un procedimiento almacenado

Esta solución debe ser tenida en cuenta si tenemos ya procedimientos en nuestro SGBD que resuelven ciertas funcionalidades, con buen rendimiento y que evitan la repetición de código.



8.2 Uso avanzado de Criteria API para actualizaciones y borrados

En la misma problemática, respecto a la cuestión del rendimiento, nos encontramos con las actualizaciones y borrados desde JPA, puesto que pueden acabar realizándose estas operaciones de una en una, degradando el rendimiento de nuestra aplicación.

En JPQL se han mencionado ya las soluciones alternativas de Bulk UPDATE y DELETE, con sus pros y contras. En el caso de Criteria API desde su versión 2.1 se han incluido soluciones similares con sentencias CriteriaUpdate y CriteriaDelete.

Ambas soluciones se utilizan de forma muy similar a como se construyen consultas con Criteria API. En el siguiente ejemplo (ver Código 57) se muestra un caso de UPDATE resuelto con CriteriaUpdate.

En el ejemplo se modifica el apellido de todos los autores (concatenando un texto prefijado) con id >= 3.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaUpdate<Author> update = cb.createCriteriaUpdate(Author.class); // crear update

Root<Author> a = update.from(Author.class); // establece la clase raíz a actualizar

// establece la actualización y cláusula where utilizando metaclasses
update.set(Author_.firstName, cb.concat(a.get(Author_.firstName), " - <actualizado>"));
update.where(cb.greaterThanOrEqualTo(a.get(Author_.id), 3L));

Query q = em.createQuery(update);
q.executeUpdate(); // ejecuta la actualización
```

Código 57: Ejemplo de CriteriaUpdate con Criteria API 2.1

El caso DELETE se resuelve de forma similar utilizando una CriteriaDelete obtenido de la invocación de createCriteriaDelete, sobre el EntityManager. Se muestra un ejemplo a continuación en el Código 58.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaDelete<Order> delete = cb.createCriteriaDelete(Order.class); // crear delete

Root e = delete.from(Order.class); // establece la clase raíz a borrar

// establece la cláusula where
delete.where(cb.lessThanOrEqualTo(e.get("amount"), amount));

Query q = em.createQuery(delete);
q.executeUpdate(); // ejecuta el borrado
```

Código 58: Ejemplo de CriteriaDelete con Criteria API 2.1

Al igual que ocurría con las operaciones *bulk*, con JPQL, debemos ser cuidadosos, puesto que estas operaciones son ejecutadas directamente contra el SGBD. Así pues el contexto de persistencia no es sincronizado (y el bloqueo optimista, que veremos en la última sección, tampoco funcionará, obligando a actualizar la columna de versión, también en la sentencia UPDATE). Tampoco se actualiza la caché.

9. Resumen

En esta sección se ha presentado el problema básico de desajuste entre las consultas tradicionales en bases de datos y las consultas con JPA. Como solución, se ha revisado un lenguaje particular de consultas como JPQL y por otro lado, se incluye una nueva solución utilizando metaclasses, que modelan consultas SQL con Criteria API. Se ha finalizado presentando pros y contras de ambas opciones.



10. Glosario

JPQL: Java Persistence Query Language.

11. Bibliografía

[Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol. ***Pro JPA 2. A definitive guide to mastering the Java Persistence API (2013)*** Apress. 2nd edition.

[Keith et al., 2018] Mike Keith, Merrick Schincariol, Massimo Nardone. ***Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence APIs.*** (2018) Apress. 3rd edition.

[Oracle, 2017] The Java EE 8 Tutorial (2017). Part VIII. Persistence. Disponible en <https://javaee.github.io/tutorial/toc.html>

[Oracle, 2014] The Java EE 7 Tutorial (2014). Part VIII. Persistence. Disponible en <http://docs.oracle.com/javaee/7/tutorial/>

12. Recursos

Especificaciones públicas:

[Oracle, 2013] JSR 338: JavaTM Persistence API, Version 2.1 (2013). Version: Final Release http://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf

[Oracle, 2017] JSR 338: JavaTM Persistence API, Version 2.2 (2017) Version: Maintenance Release http://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf

Bibliografía complementaria:

Java Persistence. (2021, September 30). *Wikibooks, The Free Textbook Project*. Retrieved 08:21, April 9, 2022 from https://en.wikibooks.org/w/index.php?title=Java_Persistence&oldid=3992576



Licencia

Autores: Raúl Marticorena & Jesús Maudes & Ana Serrano & Óscar Pérez

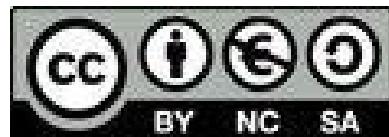
Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2022



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <http://creativecommons.org/licenses/by-nc-sa/4.0/>





Grado en Ingeniería Informática

APLICACIONES DE BASES DE DATOS

TEMA 5

Sección 5. JPA: Gestor de entidades

Docentes:

Raúl Marticorena
Jesús Maudes
Ana Serrano
Óscar Pérez

Índice de contenidos

1. INTRODUCCIÓN	3
2. OBJETIVOS	3
3. GESTOR DE ENTIDADES	3
3.1 Unidad de persistencia	5
3.2 Ciclo de vida de una entidad	6
4. OPERACIONES SOBRE ENTIDADES	7
4.1 Hacer persistente	7
4.2 Encontrar	8
4.3 Actualizar	9
4.4 Eliminar	9
4.5 Desvincular	10
4.6 Mezclar	10
4.7 Sincronizar	11
4.8 Equivalencia DML/SQL	11
4.9 Clases de utilidad	12
4.10 Orden de las operaciones	13
5. ANEXO 1. INTERCEPTORES	14
5.1 EntityListener	14
6. ANEXO 2. VALIDACIÓN	16
6.1 Características generales	16
6.2 Integración con JPA	18
7. RESUMEN	18
8. GLOSARIO	18
9. BIBLIOGRAFÍA	19
10. RECURSOS	19

Introducción

En la siguiente sección se introduce el concepto del gestor de entidades. Básicamente se encarga de gestionar, de manera transparente, las acciones necesarias para establecer el **puente** entre el modelo de objetos en memoria y las tablas en la base de datos.

Objetivos

- Conocer el concepto de gestor de entidades.
- Conocer el ciclo de vida de una entidad.
- Describir las operaciones sobre entidades.
- Auditlar las operaciones sobre las entidades.
- Conocer la API de validación.

Gestor de entidades

En primer lugar debemos señalar que un **gestor de entidades** (*Entity Manager* o EM) está ligado a un **contexto de persistencia** (*Persistence Context* o PC). Este contexto juega el papel similar al concepto de sesión en entornos web o de conexión con una base de datos. En dicho contexto, y a través del gestor de entidades, se realizarán las operaciones CRUD (*Create-Read-Update-Delete*) básicas sobre las entidades.

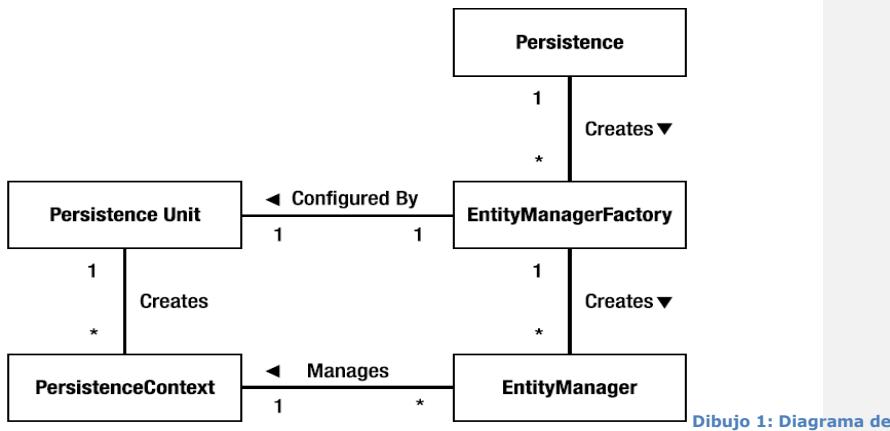
Los contextos de un gestor de entidades pueden estar:

1. Gestionados por el contenedor, propagado al resto de componentes a través de una transacción JTA (Java Transaction API). Esto típicamente en servidores de **aplicaciones JEE** (e.g. GlassFish, JBoss, etc). Esta solución está **fuerza del ámbito de esta asignatura**.
2. Gestionados por la aplicación. No se propaga y el ciclo de vida es gestionado por la propia aplicación. Típicamente en **aplicaciones JSE** como las que **son objetivo de esta asignatura**.

El gestor de entidades se obtiene a través de la interfaz `EntityManagerFactory` (EMF) (siguiendo el patrón de diseño **FactoryMethod**). La fábrica toma una configuración definida en una **unidad de persistencia** (*Persistence Unit* o PU) y permite diferenciar entre distintas fábricas. La asociación entre PU y EMF es uno a uno.

A partir de dicha unidad de persistencia (PU) se establece el contexto (PC), como el conjunto de entidades gestionadas por el gestor (EM) obtenido a partir de la interfaz fábrica (EMF).

El diagrama de clases correspondiente a la arquitectura descrita, es el mostrado en el Dibujo 1.



Dibujo 1: Diagrama de clases principales en la gestión de entidades

A continuación se muestra un ejemplo con código simplificado (ver Código 1) donde se utilizan los conceptos previamente descritos.

```

package examples.client;

import java.util.Collection;
// JPA imports
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import examples.model.Employee;
import examples.model.EmployeeDAO;

public class EmployeeClient {

    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();
        EmployeeDAO employeeDAO = new EmployeeDAO(em);

        // operations on employees using a DAO
        // ...

        // close the EM and EMF when done
        em.close();
        emf.close();
    }
}
  
```

Código 1: Ejemplo básico de uso de gestor de entidades

En este ejemplo se pueden ver en código los conceptos de persistencia, fábrica, gestor de entidades, quedando en un segundo plano menos evidente el uso de la unidad de persistencia y el contexto asociado (nombre del mismo e.g., "EmployeeService").

Unidad de persistencia

La unidad de persistencia permite definir la **configuración** de la base de datos a utilizar, así como los parámetros de configuración del proveedor de persistencia. Permite configurar y enganchar los **dos elementos básicos en JPA: el framework y el SGBD**. Es en este punto, donde realmente se toma una decisión en cuanto a qué soluciones concretas de implementación se escogen (e.g. Hibernate con Oracle).

Además en la unidad de persistencia se indican el conjunto de entidades gestionadas por el EM. Dicha configuración se incluye en un fichero con nombre `persistence.xml`, que es la **pieza clave para el correcto funcionamiento de JPA**.

A continuación vamos a ver dos ejemplos de configuración. En el primer caso (ver Código 2) se configura la PU en un servidor de aplicaciones¹, con proveedor Hibernate y utilizando un recurso JDBC definido en el servidor (en concreto un *pool* de conexiones que se localiza con JNDI). Esta solución con fuente JTA y con JDNI es habitual con servidores de aplicaciones. La biblioteca de clases con entidades persistentes a gestionar se indica con un fichero `.jar` y algunas clases adicionales (`.class`).

```
<persistence>
    <persistence-unit name="OrderManagement">
        <provider>org.hibernate.ejb.HibernatePersistenceProvider</provider>
        <description>This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            Therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

Código 2: Ejemplo de fichero `persistence.xml` en un servidor de aplicaciones JEE

En el segundo ejemplo (ver Código 3) se configura la PU para una aplicación JSE (**nuestro caso en esta asignatura**), con proveedor *Hibernate* y dando la información adicional para la cadena de conexión con la base de datos (*Derby* en este ejemplo). Las clases con entidades a gestionar se indica solo con ficheros `.class`. La gestión de transacciones en este caso **no** es gestionada por el contenedor (servidor de aplicaciones) sino que se realiza de modo local, y así se indica con el valor `transaction-type="RESOURCE_LOCAL"`.

```
<persistence>
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistenceProvider</provider>
        <class>examples.model.Employee</class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="javax.persistence.jdbc.user" value="APP"/>
            <property name="javax.persistence.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

Código 3: Ejemplo de fichero `persistence.xml` en una aplicación JSE

La raíz de la unidad de persistencia viene determinada por el fichero `.jar` o directorio que contenga el directorio `META-INF` con un fichero `persistence.xml`. El ámbito de la unidad de persistencia viene determinado por dicha raíz e identificada por un nombre único.

¹Como ya se ha indicado, el uso de servidores de aplicaciones excede el ámbito de la asignatura, pero se muestra por considerarse de interés.

Ciclo de vida de una entidad

Las entidades en JPA tienen un **ciclo de vida asociado**. Es decir, su estado persistente va a ir cambiando según se realicen ciertas operaciones sobre las entidades. La relación con el contexto de persistencia viene ligada por dicho estado. Los posibles estados son:

- **New**
 - La entidad se ha creado, no tiene identidad persistente (es un objeto Java sin más) y no está asociada a un contexto.
- **Managed**
 - Gestionada. Tiene identidad persistente y está asociada a un contexto.
- **Detached**
 - Desvinculada. Tiene identidad persistente pero no está asociada a un contexto.
- **Removed**
 - Eliminada. Tiene identidad persistente, está asociada a un contexto y se ha planificado su eliminación de la base de datos.

El diagrama de estados UML completo, con su transiciones, se muestra en la Ilustración 1.

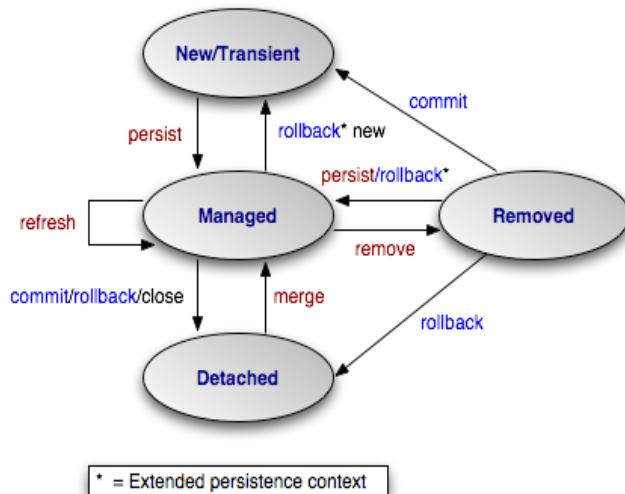


Ilustración 1: Diagrama de estado con el ciclo de vida de una entidad persistente

La transición entre estados viene dada por la invocación a los métodos del EntityManager y de la transacción vinculada, con sus operaciones `commit` y `rollback`. La figura muestra el cambio de estado en contextos de transacción, pero en contextos extendidos o gestionados por la aplicación (como es nuestro caso) puede haber alguna diferencia (e.g., tras un `commit` los objetos pueden seguir en estado `Managed`).

Operaciones sobre entidades

A continuación vamos a ir describiendo las distintas operaciones a realizar sobre las entidades persistentes. Dichas operaciones provocan la transición entre estados vista anteriormente, y modifica la gestión que sobre ellas realiza el EM.

Las operaciones se aplican fundamentalmente a través del gestor de entidades y sus métodos.

Hacer persistente

Una nueva entidad pasa a ser persistente y gestionada a través del método `persist`. Bien directamente por la invocación del método, o indirectamente, por la aplicación en cascada a través de entidades relacionadas con el valor de `cascade=PERSIST` o `cascade=ALL`.

La entidad persistente, inicialmente está en memoria, y es sólo cuando se realiza o invoca al método `flush` (a través del EM) o con el `commit` de la transacción, cuando se vuelve su estado en la base de datos.

Dependiendo de su estado previo se pueden dar distintas situaciones:

- Si previamente ya estaba gestionada (estado `managed`) se ignora la operación.
- Si previamente estaba borrada (estado `removed`) pasa a estar gestionada (`managed`).
- Si previamente estaba desvinculada (estado `detached`) se lanza una excepción `IllegalStateException` o bien la transacción fallará al hacer `flush` o `commit`.

Resumiendo, la entidad no se incorpora a la base de datos hasta que no se realiza una invocación al método `flush` o `commit`.

Encontrar

Aunque en JPA existen mecanismos de consulta muy avanzados (se verá en posteriores secciones del tema), existen unos mecanismos básicos para localizar y encontrar entidades.

El método fundamental es `find`, que permite encontrar la entidad por clave primaria. Es la operación preferente puesto que está optimizada, usando la caché de datos. Devuelve una entidad gestionada (`managed`) salvo que no haya contexto transaccional (en tal caso devuelve una entidad `detached`).

Como solución alternativa, y en caso de que sólo se necesite acceder al valor de la clave primaria, se puede utilizar el método `getReference`. Este método realmente obtiene un *proxy* (consultar el patrón de diseño **Proxy**²). Esto es útil en relaciones `@OneToOne` o `@ManyToOne` creando la entidad origen con la clave foránea, sin cargar por completo la entidad destino.

A continuación se muestran un par de ejemplos del uso de los métodos de búsqueda (ver Código 4 y Código 5).

²Una referencia al uso del *proxy* para la carga perezosa se encuentra en <https://martinfowler.com/eaaCatalog/lazyLoad.html>.

```
Department dept = em.find(Department.class, 30);
// managed department object...
Employee emp = new Employee(); // new entity object
emp.setId(53);
emp.setName("Peter");
emp.setDepartment(dept);
dept.getEmployees().add(emp);
em.persist(emp); // from new to managed
```

Código 4: Encontrar entidad con `find`

```
Department dept = em.getReference(Department.class, 30);
// using a proxy
Employee emp = new Employee(); // new entity object
emp.setId(53);
emp.setName("Peter");
emp.setDepartment(dept); // using just the proxy
dept.getEmployees().add(emp); // recover data from department
em.persist(emp); // from new to managed
```

Código 5: Encontrar entidad con `getReference`

Actualizar

La actualización en JPA es transparente: **no hay método asociado**. Se modifica el estado del objeto en memoria (métodos `setter` típicamente) y posteriormente al realizar `flush` o `commit` se vuelcan los cambios sobre la base de datos.

En el siguiente ejemplo podemos observar dicho funcionamiento (ver Código 6).

```
// Begin transaction
em.getTransaction().begin();
Employee employee = em.find(Employee.class, 1);
employee.setNickname("Joe the Plumber"); // transparent update
// End (Commit) transaction
em.getTransaction().commit();
```

Código 6: Actualización de una entidad

Se obtiene una entidad gestionada con `find`. A continuación se aplican métodos `set`, modificando su estado (pasando a estar "sucia" la entidad) para finalmente, al realizar el `commit` de la transacción, realizar los correspondientes `UPDATE` en la base de datos. Si se realiza un `rollback`, simplemente los cambios realizados en memoria son descartados.

Eliminar

Una entidad gestionada se elimina con `remove`. Bien directamente por la invocación del método o indirectamente, por la aplicación en cascada a través de entidades relacionadas con el valor de `cascade=REMOVE` o `cascade=ALL`. El acceso posterior a sus valores está indefinido.

Dependiendo de su estado previo se pueden dar distintas situaciones:

- Si previamente ya estaba borrada (estado `removed`) se ignora la operación.
- Si previamente estaba desvinculada (estado `detached`) se lanza una excepción `IllegalStateException` o bien la transacción fallará al hacer `flush` o `commit`.

La entidad se borra realmente al cierre de la transacción con flush o commit.

Sin embargo nos podemos encontrar con ciertos problemas debido a las restricciones de borrado. En el siguiente ejemplo (ver Código 7) vemos que no se puede borrar la plaza de aparcamiento, puesto que se viola una restricción de clave foránea (el empleado apuntaría a una plaza que no existe). En la segunda parte del ejemplo, se resuelve el problema estableciendo explícitamente la referencia en empleado a nulo.

```
// OneToOne between Employee and ParkingSpace
Employee emp = em.find(Employee.class, empId);
ParkingSpace ps = emp.getParkingSpace();
em.remove(ps); // ERROR foreign key constraint

// OneToOne between Employee and ParkingSpace
Employee emp = em.find(Employee.class, empId);
ParkingSpace ps = emp.getParkingSpace();
// set null in "foreign key"...
// avoid problem with foreign key constraint
emp.setParkingSpace(null);
em.remove(ps);
```

Código 7: Problema de borrado con restricción y solución

En las relaciones @OneToMany, debemos ser conscientes de que el uso de cascade=CascadeType.REMOVE o cascade=CascadeType.ALL, implican un **borrado de las entidades “a varios” de una manera poco óptima** (realiza más operaciones de las esperadas, incluyendo borrados uno a uno), aunque sea muy cómodo.

En el caso de las relaciones @ManyToMany se desaconseja el uso con cascade=CascadeType.REMOVE o cascade=CascadeType.ALL, puesto que **borra no solo filas en la tabla intermedia (join table), sino también las entidades en el otro extremo de la relación**. Si además en el otro extremo también se utiliza cascade, se puede finalizar **borrando muchos más datos de los esperados** con una explosión de operaciones SQL.

Como primera alternativa se sugiere el uso del **borrado de los elementos de la colecciones** (no el remove del EntityManager) para que se genere el código SQL correspondiente, solo al borrado de las filas en la join table.

La segunda alternativa es utilizar **BULK DELETE con los pros y contras** ya vistos. Se puede consultar una explicación amplia sobre este problema en <https://www.thoughts-on-java.org/avoid-cascadetype-delete-many-assocations/>.

Desvincular

Se puede desvincular una entidad del contexto de persistencia a partir de las siguientes situaciones:

- Al cerrar el contexto de persistencia (Ej: em.close());
- Invocando al método clear() de un EntityManager.
- En una transacción, con contexto de persistencia con ámbito de transacción, al realizar el commit.
- Cuando ocurre un rollback, todas la entidades asociadas con la transacción se desvinculan.
- Usando el método detach() sobre una entidad (y sus relaciones con CASCADE o ALL).
- En un *stateful session bean* con un contexto de persistencia, al ser eliminado.
- Al serializar una entidad.

Estos **dos últimos casos** quedan fuera del ámbito de la asignatura.

Por otro lado, nos podemos encontrar con algún problema adicional: si hay carga perezosa (LAZY) y se intenta acceder a través de una entidad desligada, el comportamiento es dependiente del proveedor (incluyendo el posible lanzamiento de excepciones).

Por último, recordad que una vez desvinculada la entidad ya no podemos aplicar ciertas operaciones sobre la entidad como bloqueos (`lock`), navegación, etc.

Mezclar

Se puede realizar la combinación o mezcla con el método `merge`. Una entidad no gestionada (`detached`) pasa a ser de nuevo gestionada (`managed`). Pero, ojo, se devuelve (se genera) una **nueva entidad** sobre la que habrá que realizar los cambios necesarios (ver Código 8).

```
public void updateEmployee(Employee emp) {  
    // Merge a detached entity, modified in other layer  
    Employee managedEmp = em.merge(emp);  
    // using the returned object... update the date  
    managedEmp.setLastAccessTime(new Date());  
}
```

Código 8: Mezcla de una entidad

Dependiendo de su estado previo se pueden dar distintas situaciones:

- Si se aplica a una entidad nueva se genera una nueva entidad gestionada (`managed`) con la copia de los valores.
- Si se aplica a una entidad gestionada (`managed`) se ignora, aunque se aplica en cascada.
- Si se aplica a una entidad eliminada (`removed`) se lanza una excepción `IllegalArgumentException`.

Sincronizar

La sincronización **hacia la base de datos (de memoria a BD)** se realiza con los métodos `flush` y `commit`. Si la entidad gestionada tiene una relación bidireccional con otra entidad gestionada, se hacen persistentes los datos en base al propietario de la relación.

El método `flush` se usa en pasos intermedios de la transacción para forzar la sincronización entre el modelo de objetos y los datos en las bases de datos (por defecto se suelen almacenar en una caché todas las operaciones SQL hasta que se hace un `commit`, salvo que se ejecute `flush`). También fuerza la sincronización con entidades relacionadas con `cascade=PERSIST` o `cascade=ALL`.

Si lo que se quiere es realizar la sincronización **desde la base de datos (de BD a memoria)**, refrescando las instancias del gestor de entidades, se usa el método `refresh`. Solo aplicable a entidades gestionadas. Si se aplica sobre una entidad desvinculada se provoca el lanzamiento de la excepción `IllegalArgumentException`. Utilizado principalmente con contextos extendidos, no ligados a una única transacción (de vida breve), sino a transacciones largas (con bloqueo optimista³).

En el siguiente ejemplo, se muestra el uso de dicho método (ver Código 9). Si el tiempo transcurrido desde la última carga de datos (o refresco) es mayor que una cierta cota de tiempo, se "refresca" de nuevo el estado del objeto, *resetearlo* la marca de tiempo de carga.

³Los bloqueos optimistas se explicarán en otra sección posterior.

```

private void refreshEmployeeIfNeeded() {
    if ((System.currentTimeMillis() - loadTime) > REFRESH_THRESHOLD) {
        em.refresh(emp); // REFRESH!!!
        loadTime = System.currentTimeMillis();
    }
}

```

Código 9: Sincronización desde la BD con refresh

Equivalencia DML/SQL

A continuación se presenta una **tabla resumen de las operaciones JPA y su correspondiente operación SQL y CRUD**. Se incluyen también las operaciones de JPQL equivalentes.

Operación/Concepto JPA	Comando SQL	Operación CRUD
1) executeQuery (JPQL) 2) (Criteria API)	SELECT	READ
persist*	INSERT	CREATE
1) (transparente modificando el estado del objeto)* 2) executeUpdate (JPQL) 3) merge (en diferido)	UPDATE	UPDATE
1) remove* 2) executeDelete (JPQL)	DELETE	DELETE

*Suponiendo que se aplica flush o commit.

Clases de utilidad

Finalmente, indicar que para consultar el estado de las entidades se proporciona una clase de utilidad PersistenceUtil con un par de métodos:

- isLoaded(Object) consulta si las propiedades no perezosas de la entidad están cargadas.
- isLoaded(Object, String) consulta si la propiedad coincidente en nombre está cargada.

En el siguiente código se dan un par de ejemplos de uso:

```

Persistence.getPersistenceUtil().isLoaded(em.getReference(Employee.class, 42));
Persistence.getPersistenceUtil().isLoaded(em.find(Employee.class, 42), "phoneNumbers");

```

Código 10: Ejemplos de uso de isLoaded

También existe la clase PersistenceUnitUtil con mismos métodos, y tienen un menor coste al estar asociados a la fábrica. Incluye un método adicional getIdentifier para consultar el identificador de la entidad.

```

public List<Object> getEntityIdentifiers(List<T> entities) {
    PersistenceUnitUtil util = emf.getPersistenceUnitUtil();
    List<Object> result = new ArrayList<Object>();
    for (T entity : entities) {
        result.add(util.getIdentifier(entity));
    }
    return result;
}

```

Código 11: Ejemplos de uso de consulta de identificador de entidad

Orden de las operaciones

Aunque en una transacción con JPA se realizan (o programan) las **operaciones en un cierto orden sobre los objetos**, se puede comprobar, auditando las operaciones SQL, que **el orden de ejecución posterior** por parte del **framework**, **NO siempre es coincidente**.

El framework “**acumula y retrasa**” las operaciones pendientes, incluso con un nuevo orden, hasta que finalmente se realiza el `commit` (o bien se fuerza en pasos intermedios de la transacción, con `flush`).

En concreto, *frameworks* como *Hibernate*, indican explícitamente en su documentación que el orden será finalmente:

1. inserciones (insert)
2. actualizaciones (updates)
3. borrados de colecciones de elementos (delete)
4. inserciones de colecciones de elementos (insert)
5. borrados (delete)

Este orden garantiza preservar la integridad referencial (además de realizar cambios sobre la caché de nivel 1). El retraso en el envío de varias operaciones se beneficia también de mecanismos como *JDBC Batching*, para reducir el número de viajes de ida y vuelta a la base de datos.

Si en algún caso se quiere **forzar** a enviar todas las operaciones SQL **pendientes hasta el momento**, es necesario **utilizar explícitamente** el método `flush`. Aunque estos cambios no serán visibles para el resto de transacciones hasta que finalmente se haga el `commit`, como ya ocurría en una transacción desde la consola con SQL*Plus, SQLDeveloper, PL/SQL o JDBC (suponiendo que estamos con nivel de aislamiento READ COMMITTED o SERIALIZABLE).

Para asegurar **consistencia en las consultas** (leer los datos modificados por la propia transacción) se suele trabajar en modo `flush-before-query`, donde se fuerza a realizar un `flush` antes de la consulta. En concreto, en la especificación de JPA tenemos dos modos definidos (propiedad `FlushModeType`):

- AUTO: el proveedor es responsable de asegurar que todas las actualizaciones de la entidades en el contexto de persistencia que podrían afectar al resultado de la consulta son visibles (no obliga a resolverlo de una manera específica). Este es el **valor por defecto**, garantizando que la transacción consulta los valores previamente modificados.
- COMMIT: se realiza un `flush` implícito con el `commit`, sin especificarse si el efecto de las actualizaciones a las entidades es visible o no a las consultas (no se asegura).

Aunque este es el comportamiento indicado en la especificación, como suele ser habitual, cada *framework* comercial puede realizar algunas variaciones (Ej: Hibernate incluye modos adicionales como `ALWAYS` y `MANUAL`).

De nuevo es **muy importante** remarcar la importancia de auditar las operaciones SQL realizadas por el *framework*, para detectar errores y optimizar rendimientos, como ya se indicó en secciones previas.

Comentado [Autor des1]: Esta estrategia se denomina transactional write behind.

Comentado [Autor des2]: Ver <https://vladmihalcea.com/hibernate-facts-knowing-flush-operations-order-matters/>

Anexo 1⁴. Interceptores

En JPA se incluye el concepto de inclusión de **interceptores**. Estos permiten interceptar **eventos del ciclo de vida de las entidades** y proporcionar **métodos de retrollamada** (*callback methods*) a ejecutar en cada caso. Para ello, los métodos interceptores deben cumplir una serie de condiciones:

- No estáticos.
- No finales.
- Sin argumentos.
- Retorno void.

El método en cuestión se anota con alguno de los siguientes valores, en función del momento del ciclo de vida en el que queremos que sean invocados: @PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, @PostRemove y @PostLoad (se pueden aplicar varias al mismo método).

Por otro lado se restringen ciertas operaciones a realizar en estos métodos: invocar métodos del EntityManager u obtener consultas de él.

Los interceptores se pueden definir en métodos de la propia entidad (ver Código 12).

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @Transient private long syncTime;
    // ...
    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() {
        syncTime = System.currentTimeMillis();
    }
    public long getCachedAge() {
        return System.currentTimeMillis() - syncTime;
    }
    // ...
}
```

Código 12: Interceptores definidos en una entidad Employee

EntityListener

Como solución alternativa a la definición de los métodos en la entidad, se plantea la construcción de clases funcionales (sin estado) que implementan métodos de retrollamada (*callback methods*) sobre las entidades.

Estas clases tienen un constructor sin argumentos, sin atributos (sin estado) y con una firma específica en los métodos:

- No estáticos.
- No finales.

⁴Los anexos de estas secciones contienen material adicional que **NO entra como materia de evaluación**, pero se deja disponible para su consulta posterior, por si fuera de utilidad para los alumnos.

- Reciben como argumento de entrada `Object`, tipo entidad, una superclase de la entidad o una interfaz implementada por la entidad.

- Retorno `void`.

Una vez construidos estos oyentes (ver Código 13), se registran en la entidad mediante anotaciones `@EntityListeners` (ver Código 14).

```
public class NameValidator {
    static final int MAX_NAME_LEN = 40;

    @PrePersist
    public void validate(NamedEntity obj) {
        if (obj.getName().length() > MAX_NAME_LEN)
            throw new ValidationException("Id.out of range");
    }

    public class EmployeeDebugListener {
        @PrePersist
        public void prePersist(Employee emp) {
            System.out.println("Persist on employee id: " + emp.getId());
        }

        @PreUpdate    public void preUpdate(Employee emp) { ... }
        @PreRemove   public void preRemove(Employee emp) { ... }
        @PostLoad    public void postLoad(Employee emp) { ... }
    }
}
```

Código 13: Ejemplo de dos oyentes, validación y debug con log

En este ejemplo, la entidad también es oyente de sus eventos, puesto que en el método necesita acceder al estado de la entidad, no accesible para los otros oyentes (ver uso de la variable `syncTime`).

```
@Entity
@EntityListeners({EmployeeDebugListener.class, NameValidator.class})
public class Employee implements NamedEntity {
    @Id private int id;
    private String name;
    @Transient private long syncTime;
    public String getName() { return name; }
    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() {
        syncTime = System.currentTimeMillis();
    }
    public long getCachedAge() {
        return System.currentTimeMillis()
               - syncTime;
    }
    // ...
}

public interface NamedEntity {
    public String getName();
}
```

Código 14: Ejemplo de registro de oyente

Al utilizar mecanismos de herencia entre las entidades se establece el siguiente orden de invocación para los oyentes:

- Los métodos de retrollamada se invocan en orden de lo más general a lo más particular, primero los de las superclases y después los de las subclases, de forma similar al mecanismo de invocación de constructores en las jerarquías de herencia.

- Los oyentes añadidos con anotaciones, también se invocan en orden de lo más general a lo más particular, primero los de las superclases y después los de las subclases. Se agregan siempre, si se quiere excluir en un descendiente se debe utilizar la anotación `@ExcludeSuperclassListeners`.

Anexo 2. Validación

Aunque no sea una parte integral de JPA, se puede incluir el uso de la API de validación (Bean Validation 1.0). Permite la inclusión de validaciones sobre los datos, de forma similar a las restricciones en bases de datos.

Características generales

Básicamente utiliza mecanismo bien conocidos como anotaciones y validadores, aplicados sobre la ejecución de métodos `get` y `set`. Incluye además un conjunto de restricciones predefinidas (paquete `javax.validation.constraints`).

La lista de restricciones predefinidas (anotaciones) se incluye a continuación⁵:

Constraint	Attributes ³	Description
<code>@Null</code>		Element must be null
<code>@NotNull</code>	Element must not be null	
<code>@AssertTrue</code>	Element must be true	
<code>@AssertFalse</code>	Element must be false	
<code>@Min</code>	<code>long value()</code>	Element must have a value greater than or equal to the minimum
<code>@Max</code>	<code>long value()</code>	Element must have a value less than or equal to the maximum
<code>@DecimalMin</code>	<code>String value()</code>	Element must have a value greater than or equal to the minimum
<code>@DecimalMax</code>	<code>String value()</code>	Element must have a value less than or equal to the maximum
<code>@Size</code>	<code>int min()</code> <code>int max()</code>	Element must have a value between the specified limits
<code>@Digits</code>	<code>int integer()</code> <code>int fraction()</code>	Element must be a number within the specified range
<code>@Past</code>		Element must be a date in the past
<code>@Future</code>		Element must be a date in the future
<code>@Pattern</code>	<code>String regex()</code> <code>Flag[] flags</code>	Element must match the specified regular expression (Flags offer regular expression settings)

Aunque la invocación de las validaciones puede ser implícita, también se permite que sea explícita a través de `javax.validation.Validator`. Se utilizan mecanismos de inyección de dependencias (entornos JEE) o por método de fabricación (entorno JSE no sólo con JPA).

El siguiente ejemplo muestra el uso de validadores de manera explícita sobre una entidad (ver Código 15).

⁵Consultar la documentación online en <http://docs.oracle.com/javaee/7/tutorial/partbeanvalidation.htm#sthref1322>

El método `validate` realiza las validaciones correspondientes, recogiendo como resultado el conjunto de las restricciones violadas.

```
{  
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();  
    Validator validator = factory.getValidator();  
}  
  
public void newEmployee(Employee emp) {  
    Set<ConstraintViolation<Employee>> constraintViolations = validator.validate(emp);  
    assertEquals("Should raise 1 constraint", 1, constraintViolations.size());  
    //...  
}
```

Código 15: Ejemplo de validación explícita sobre entidad

Cuando una restricción falla de manera irrecuperable, se lanza una excepción `ValidationException` con descendientes: `ConstraintDeclarationException`, `ConstraintDefinitionException`, `ConstraintViolationException`, `GroupDefinitionException`.

Para facilitar la gestión de las restricciones, se pueden definir **grupos**, indicando cuál o cuáles aplicar. Se definen grupos a través de interfaces y posteriormente se indican en las restricciones el grupo al que pertenecen (ver Código 16). Al aplicar el método `validate` se puede indicar qué grupos aplicar.

```
// Group definitions...  
public interface FullTime extends Default {}  
public interface PartTime extends Default {}  
  
public class Employee {  
    @NotNull // Default...  
    private int id;  
    @NotNull // Default...  
    @Size(max=40) // Default...  
    private String name;  
  
    @NotNull(groups=FullTime.class)  
    @Null(groups=PartTime.class)  
    private long salary;  
  
    @NotNull(groups=PartTime.class)  
    @Null(groups=FullTime.class)  
    private double hourlyWage;  
    // ...  
}
```

Código 16: Creación de grupos y aplicación

También se permite la aplicación de varios grupos a la vez (ver Código 17).

```
public class Employee {  
    @NotNull(groups={FullTime.class,PartTime.class})  
    private int id;  
    @NotNull(groups={FullTime.class,PartTime.class})  
    @Size(groups={FullTime.class,PartTime.class},max=40)  
    private String name;  
    // ...  
}
```

Código 17: Aplicación de varios grupos de validación

Integración con JPA

Para poder combinar el uso de la validación con JPA, es necesario activar el modo de validación:

- Bien en el fichero `persistence.xml`, añadiendo la siguiente línea:
 - <property name="javax.persistence.validation.mode" value="none" />
- Con valores (value):
 - AUTO—activar cuando un proveedor de validación esté presente en el CLASSPATH
 - CALLBACK—activar y lanzar un error si no hay proveedor de validación
 - NONE—desactivar
- Bien como propiedad `javax.persistence.validation.mode`
 - Se especifica en un Map pasado al método `createEntityManagerFactory()`
 - Con valores auto, callback o none.

La validación está integrada implícitamente con JPA en el ciclo de vida de las entidades (**no se invoca explícitamente a validate**).

Los puntos por defecto para su aplicación (grupo default) son `@PrePersist`, `@PreUpdate` y `@PreRemove`, pudiendo lanzar excepciones de tipo `javax.validation.ValidationException` (hereda de `RuntimeException`). La excepción encapsula dentro la lista de `ConstraintViolation`, con toda su información asociada.

Además cuando se usa `@Valid` sobre un atributo o propiedad, se valida también el objeto almacenado (e.g. con embebidos), pero no se validan relaciones.

Resumen

En esta sección se ha revisado el uso de los gestores de entidades, su ciclo de vida y las operaciones a realizar con el gestor de entidades. Se ha presentado el concepto de interceptores, para auditar el ciclo de vida de las entidades y se ha finalizado presentando la integración del mecanismo de validación con JPA.

Glosario

PC: contexto de persistencia (*persistence context*)

EMF: fábrica de gestores de entidades (*entity manager factory*).

EM: gestor de entidades (*entity manager*).

PU: unidad de persistencia (*persistence unit*).

Bibliografía

[Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol. **Pro JPA 2. A definitive guide to mastering the Java Persistence API (2013)** Apress. 2nd edition.

[Keith et al., 2018] Mike Keith, Merrick Schincariol, Massimo Nardone. **Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence APIs.** (2018) Apress. 3rd edition.

[Oracle, 2017] The Java EE 8 Tutorial (2017). Part VIII. Persistence. Disponible en <https://javaee.github.io/tutorial/partpersist.html>

[Oracle, 2014] The Java EE 7 Tutorial (2014). Part VIII. Persistence. Disponible en
<http://docs.oracle.com/javaee/7/tutorial/>

Recursos

Bibliografía complementaria:

Java Persistence. (2021, September 30). *Wikibooks, The Free Textbook Project*. Retrieved 08:21, April 9, 2022 from https://en.wikibooks.org/w/index.php?title=Java_Persistence&oldid=3992576

Licencia

Autores: Raúl Marticorena & Jesús Maudes & Ana Serrano & Óscar Pérez

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2022



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <http://creativecommons.org/licenses/by-nc-sa/4.0/>



Grado en Ingeniería Informática

APLICACIONES DE BASES DE DATOS

TEMA 4

Sección 6. JPA: Transacciones y
conurrencia

Docentes:

Raúl Marticorena
Mario Martínez
Pablo García

Índice de contenidos

1. INTRODUCCIÓN.....	3
2. OBJETIVOS.....	3
3. TRANSACCIONES.....	3
4. CONCURRENCIA.....	5
4.1 Bloqueos optimistas.....	6
4.2 Bloqueos pesimistas.....	8
4.3 Obtención de bloqueos.....	10
5. CACHÉ.....	11
5.1 Caché del contexto de persistencia - Nivel 1.....	12
5.2 Caché compartida (Shared Cache) – Nivel 2.....	12
6. RESUMEN.....	14
7. GLOSARIO.....	14
8. BIBLIOGRAFÍA.....	14
9. RECURSOS.....	15



1. Introducción

En este tema se concluye con un estudio más detallado del concepto de transacción en JPA, con sus características y diferencias respecto a las transacciones tradicionales con bases de datos. Por otro lado se revisan los problemas del acceso concurrente y los posibles bloqueos que pueden ser establecidos. Finalmente se revisa el concepto de caché para las entidades, las distintas *cachés* disponibles y su sincronización desde y hacia la base de datos.

2. Objetivos

- Describir la definición de transacciones en JPA.
- Conocer los tipos de bloqueo en acceso concurrente con JPA.
- Detallar las distintas cachés y su uso.

3. Transacciones

Aunque ya se han venido realizando transacciones con JPA en prácticas, se va a revisar a continuación en un mayor grado de profundidad algunas de sus particularidades.

Dado que JPA interactúa con un SGBD, a través del *framework* de persistencia, debemos tener en consideración el concepto de transacción.

La especificación de JPA, en su sección **3.4 Locking and Concurrency**, indica que el **valor por defecto esperado de aislamiento es READ_COMMITTED**. Citando dicha especificación "*las aplicaciones pueden requerir niveles de aislamiento superiores a read-committed. La configuración del nivel de aislamiento, queda fuera del ámbito de la especificación*". Por lo tanto, ese punto es dependiente de la solución particular del proveedor de persistencia.

Por poner un ejemplo concreto, en nuestro caso con *Hibernate*, se puede establecer en el fichero de configuración (*persistence.xml*), el nivel de aislamiento para la fuente de datos utilizada. Suponemos además que **no se usa una fuente JNDI externa**, con su propia configuración de aislamiento:

Ej:<property name="hibernate.connection.isolation">8</property>

donde el valor numérico es 1=DIRTY_READ, 2=READ_COMMITTED, 4=REPEATABLE_READ y 8=SERIALIZABLE.

O bien en código estableciendo las propiedades de conexión:

Ej:

```
properties.setProperty("hibernate.connection.isolation",String.valueOf(Connection.TRANSACTION_SERIALIZABLE));
```

Por otro lado, las transacciones en JPA pueden estar gestionadas:

- Por el servidor
 - Típicamente en JEE, con servidores de aplicaciones usando Java Transaction API (JTA) y la interfaz `UserTransaction` (fuera del ámbito de la presente asignatura).
- Por la aplicación
 - En aplicaciones JSE, como recurso local indicado en el fichero *persistence.xml*.



- Ej: <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL"/>
- En este segundo caso, las transacciones son manejadas a través de la interfaz EntityTransaction obtenida del gestor de entidades (EntityManager) a través del método getTransaction.
- La interfaz EntityTransaction ofrece la siguiente colección de métodos, asociados al ciclo de vida de la transacción (ver Código 1).

```
public interface EntityTransaction {
    public void begin(); // IllegalStateException without closing previous
    public void commit(); // RollbackException if it fails
    public void rollback(); // PersistenceException if it fails
    public void setRollbackOnly(); // Mark the transaction to be rolled back
    public boolean getRollbackOnly(); // Query if it has to be rolled back
    public boolean isActive();
}
```

Código 1: Interfaz EntityTransaction

El ámbito transaccional se mantiene desde que se invoca al método `begin` hasta que se ejecuta el método `commit` (éxito en la transacción) o `rollback` (fracaso en la misma). Todas las operaciones sobre entidades ejecutadas entre estos métodos se ejecutan en el ámbito de una transacción JPA. En el Código 2, se muestra un ejemplo de una típica transacción. La transacción es enteramente gestionada (su obtención, inicio y finalización) desde la aplicación JSE.

```
public class ExpirePasswords {
    public static void main(String[] args) {
        int maxAge = Integer.parseInt(args[0]);
        String defaultPassword = args[1];
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("admin");
        EntityManager em = emf.createEntityManager();
        try {
            Calendar cal = Calendar.getInstance();
            cal.add(Calendar.DAY_OF_YEAR, -maxAge);
            em.getTransaction().begin();           // Begin transaction
            // Query
            Collection expired =
                em.createQuery("SELECT u FROM User u WHERE u.lastChange <= ?1")
                    .setParameter(1, cal).getResultList();

            // Update
            for (Iterator i = expired.iterator(); i.hasNext();) {
                User u = (User) i.next();
                System.out.println("Expiring password for " + u.getName());
                u.setPassword(defaultPassword);
            }
            em.getTransaction().commit();          // Commit transaction
        }
        catch(Exception ex) {
            if (em.getTransaction().isActive()){
                em.getTransaction().rollback(); // Rollback transaction
            }
        } finally {
            em.close();
            emf.close();
        }
    }
}
```

Código 2: Ejemplo de transacción con JPA



Ante cualquier problema sobre las operaciones de persistencia (estando la transacción activa – método `isActive` devuelve `true`) , se marca la transacción para `rollback` por parte del proveedor. Se puede consultar su estado (`getRollbackOnly`) y aplicar la operación de `rollback`. También se puede fijar dicho valor desde el propio programa con el método simétrico `setRollbackOnly`.

Si se intenta hacer un `commit` sobre una transacción marcada para deshacer o bien se produce cualquier error en el `commit`, se lanzará una excepción `RollbackException`. Sin embargo si se ejecuta `rollback` y se produce algún error, se lanzará `PersistenceException`¹.

Un problema con JPA es que al realizarse el `rollback` no se asegura la preservación de una instantánea de las entidades del contexto, consistente con la base de datos: las entidades realmente se desvinculan (en estado `detached`) y reflejan el estado en el momento de ejecutarse el `rollback`. **Por lo tanto NO es equivalente a un `rollback` sobre la base de datos.**

¿Qué ocurre realmente en el ORM al producirse un `rollback`? Los cambios se deshacen en la base de datos (sí se produce un `rollback` en el SGBD) pero el contexto de persistencia se “limpia” y las entidades se desvinculan.

Así pues surge un problema con las entidades que tenemos en la actualidad. Por ejemplo, podrían tener `id` no únicos, y sería necesario generar nuevos (si además usamos atributos para el control de versión – `@Version` – también podrían pasar a ser inválidos, como veremos a continuación).

Como solución se plantea “reiniciar la transacción”, con la creación de nuevas entidades gestionadas y la copia de valores.

4. Conurrencia

Cuando se tienen varios usuarios (con sus correspondientes aplicaciones en nuestro caso) accediendo a la base de datos (a través de JPA) se producen accesos concurrentes. En concurrencia, se debe asegurar la integridad de los datos. Esto se consigue en JPA a través de un enfoque optimista (utilizando versionado) o bien pesimista (con bloqueos), de manera similar a como se estableció en la teoría del primer tema de transacciones de la asignatura.

Para ello, el proveedor de persistencia asume que la base de datos tiene²:

- bloqueos de lectura de grano fino
- bloqueos de escritura de grano grueso

En todo caso, en JPA, las **escrituras** en la base de datos **se retrasan lo máximo posible** hasta encontrarnos con un `flush` o `commit` en nuestro código, como se ha comentado en secciones anteriores

Por defecto, se utilizan bloqueos optimistas. En dichos bloqueos, es necesario mantener una columna en la base de datos y un atributo en la entidad, con el número de versión³. En JPA se introduce una nueva anotación `@Version`, utilizándose sobre el atributo correspondiente de la entidad. Este valor NO debe ser nunca gestionado por la aplicación, sino que es algo transparente a la misma, gestionado por el proveedor de persistencia.

Si hay cualquier desacuerdo entre los valores (entidad vs. filas de la tabla en la BD) se lanzará una excepción `OptimisticLockException`. Los posibles tipos para el atributo anotado con `@Version` son: `int`,

¹ Esta excepción hereda de `RuntimeException`, lo que no obliga a su captura ni propagación de manera explícita, aunque debería tenerse en cuenta para la robustez de la aplicación final.

² En el caso de Oracle ya hemos visto, en los primeros temas, que no se cumple esta suposición. Esto es muy dependiente del SGBD concreto que se utilice.

³ No todos los proveedores requieren el uso de un campo `@Version` en la entidad, pero esto es particular, y no general a JPA. Es criticable esta solución, al imponer cambios sobre la base de datos.



`Integer, long, Long, short, Short, Date y java.sql.Timestamp`. Este valor se actualiza⁴ en la BD cuando se realiza flush o commit, o bien se adquieren bloqueos de escritura.

4.1 Bloqueos optimistas

En los bloqueos optimistas se supone que la transacción actual **es la única que está cambiando "estos datos concretos"**. No adquiere bloqueos hasta que se vuelcan los cambios, al **final de la transacción. El bloqueo optimista es el valor implícito en las escrituras**.

Si en dicho momento, se detecta que los datos **han sido modificados por otra transacción**, se lanza una excepción `OptimisticLockException`. Así pues, nos encontramos en una **situación por defecto de NO bloqueos**, con un nivel de aislamiento ANSI/ISO SQL de tipo **READ COMMITTED**, donde se garantiza que se lean siempre datos cometidos por otras transacciones.

En la siguiente Ilustración 1, se muestra el efecto de utilizar el bloqueo optimista **por defecto**, usando `@Version`.

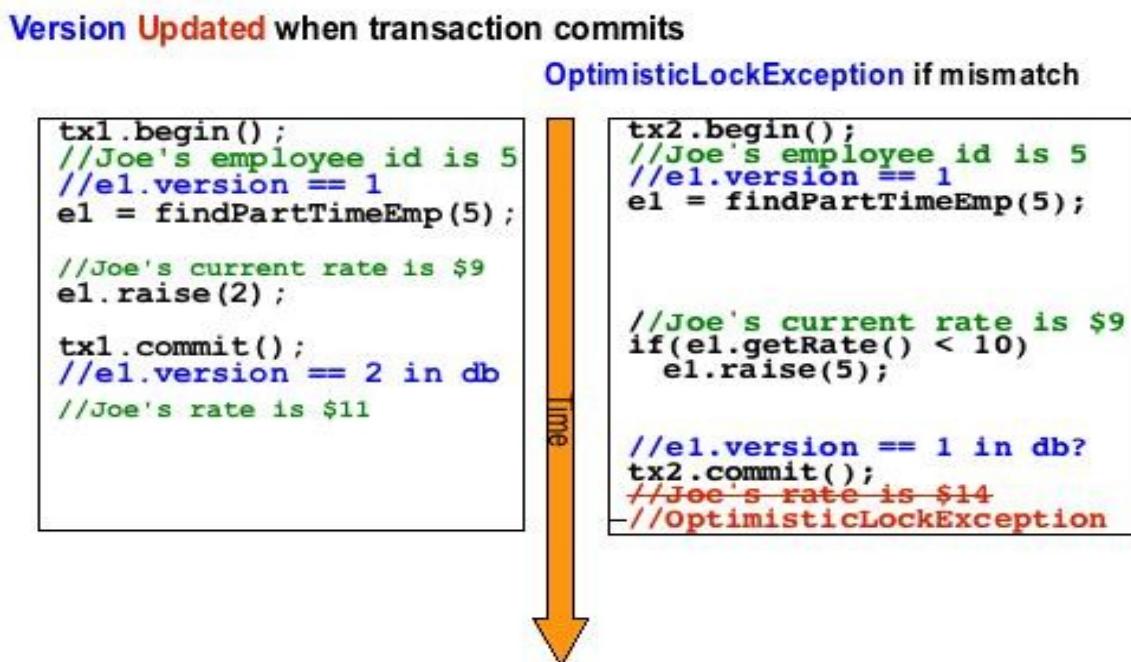


Ilustración 1: Bloqueo optimista por defecto. Ilustración extraída de [McDonald, C. 2009]

Como se puede observar, ambas transacciones obtienen el dato (el objeto) con valor de versión=1. Cuando la segunda transacción, que también modifica el dato, intenta hacer un `commit`, se comprueba que **el dato ha sido modificado previamente por la primera transacción** (el nº de versión es diferente $2 \neq 1$), y salta la excepción `OptimisticLockException`.

Si se quiere cambiar a un nivel de aislamiento **REPEATABLE READ**, asegurando que **datos leídos no son tampoco modificados en otra transacción** (si repetimos la lectura, obtendríamos la misma lectura), se puede realizar el cambio pasando a uno de los métodos de bloqueo, el parámetro `LockModeType.OPTIMISTIC`. Cambiando el tipo de bloqueo a `OPTIMISTIC`, tendríamos la situación mostrada en la Ilustración 2.

4 No se garantiza su actualización en operaciones *bulk update* por parte de todos los proveedores.



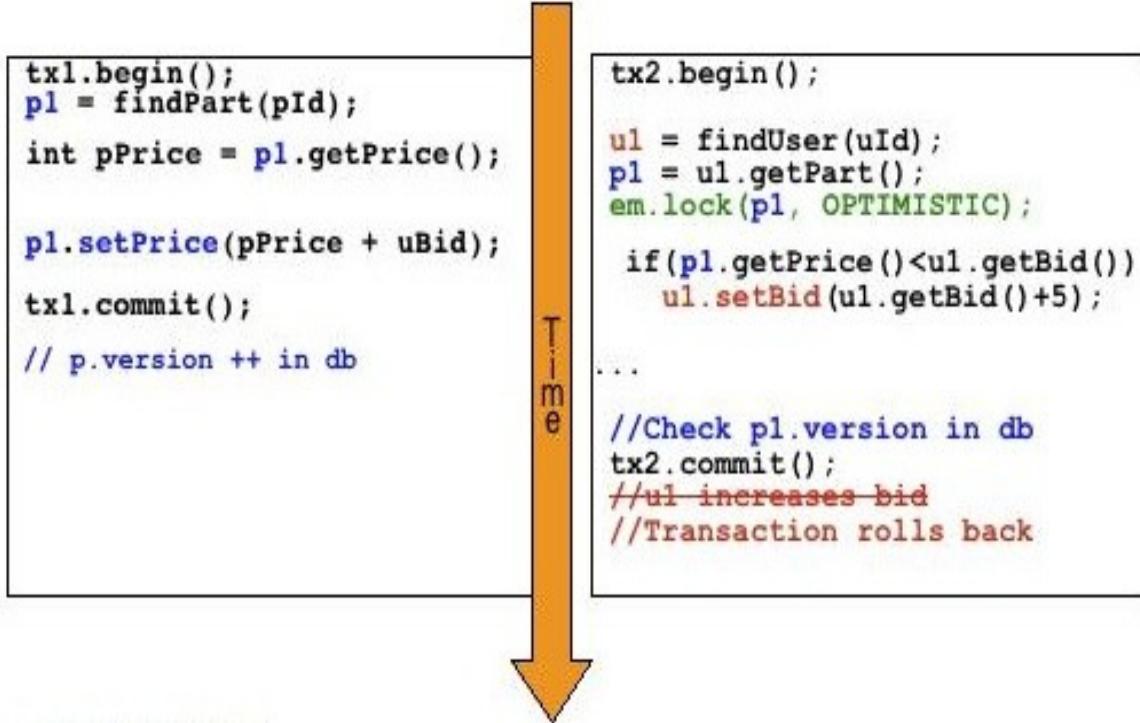


Ilustración 2: Bloqueo optimista de tipo OPTIMISTIC. Ilustración extraída de [McDonald, C. 2009]

En este segundo caso, la primera transacción actualiza el precio del producto (p1). En la segunda transacción **se lee previamente** el valor del precio de ese mismo producto (p1) y se realiza una **actualización posterior, derivada** de ese valor leído **anteriormente**.

Si se permitiese el `commit` de la segunda transacción, **se estaría realizando una actualización en base a un valor leído (versión 1) que no coincide con la versión actual (versión 2)**. En tal caso, se lanza una `OptimisticLockException` y se debe deshacer la transacción (rollback).

Debemos observar que se ha colocado un **bloqueo “optimista” explícitamente** sobre la entidad versionada (e.g. `em.lock(p1, OPTIMISTIC)`).

Finalmente, dentro de los bloqueos optimistas, podemos seleccionar **bloqueos de escritura**, donde se incrementa automáticamente el nº de versión. Para ello se pasa el valor:

`LockModeType.OPTIMISTIC_FORCE_INCREMENT.`

En la Ilustración 3, se muestra un ejemplo. En este modo se genera un **fallo de bloqueo en otra transacción, si se intenta modificar la entidad bloqueada (o bloquearla para lectura)**. Se suele utilizar para garantizar la consistencia entre entidades en una relación.



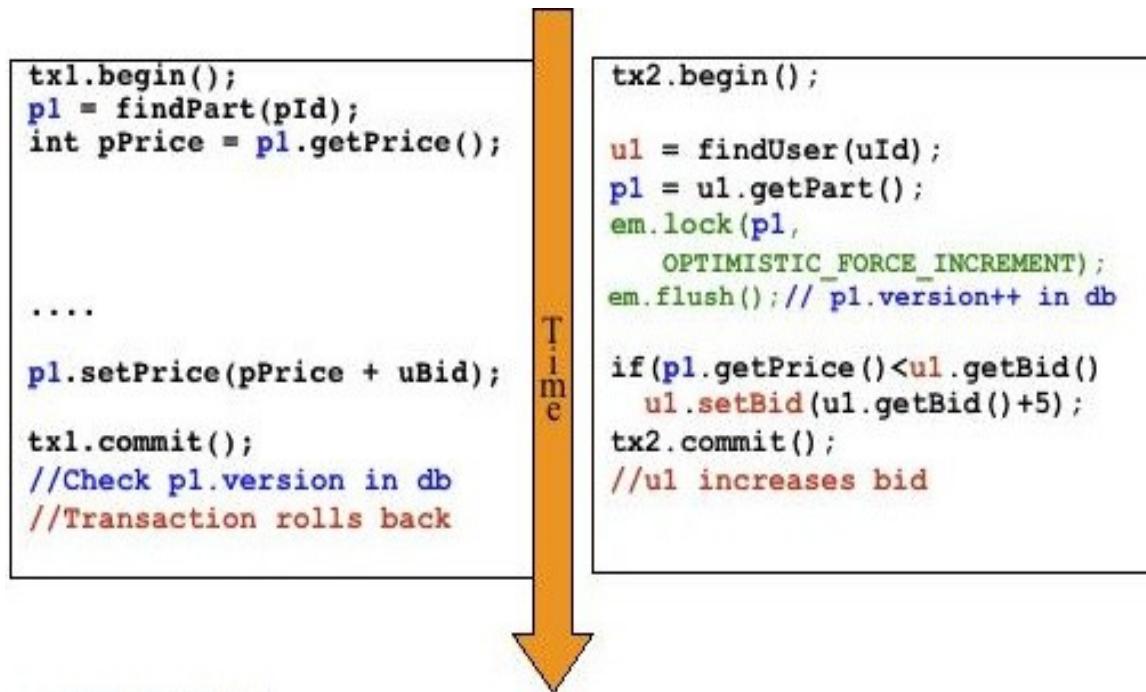


Ilustración 3: Bloqueo optimista de tipo `OPTIMISTIC_FORCE_INCREMENT`. Ilustración extraída de [McDonald, C. 2009]

La segunda transacción intenta asegurarse de que **nadie cambia el valor del producto** (`p1`). Se fuerza el envío de la modificación de la versión con `flush`. Cualquier intento de **otra transacción por modificar u obtener un bloqueo de lectura**, sobre dicho producto, lanzará una excepción `OptimisticLockException` en la otra transacción y deberá realizar un `rollback`.

4.2 Bloqueos pesimistas

Con el bloqueo pesimista se obtiene el bloqueo **en el mismo momento del acceso, de manera síncrona y exclusiva a nivel de base de datos**, asegurando que ninguna otra transacción puede obtener bloqueos sobre el objeto.

Se asegura que las transacciones no actualizan la misma entidad al mismo tiempo, **limitando el acceso concurrente**. Deben ser usados con moderación puesto que pueden generar **problemas de baja escalabilidad, tiempos de espera largos, deadlocks, etc.**

Los modos definidos (en `LockModeType`) en JPA son tres:

1. `PESSIMISTIC_WRITE`
 - Evita que otras transacciones obtengan bloqueos `PESSIMISTIC_READ` o `PESSIMISTIC_WRITE`, impidiendo lecturas, modificaciones o borrados en otras transacciones. Se corresponde con un bloqueo exclusivo. En la Ilustración 4 e Ilustración 5, se muestran un par de ejemplos de uso de estos bloqueos.



```
//Read and lock:  
  
Part p = em.find(Part.class, pId, PESSIMISTIC_WRITE);  
// update (p already locked)  
int pAmount = p.getAmount();  
p.setAmount(pAmount - uCount);
```

Locks longer,
could cause
bottlenecks

Ilustración 4: Bloqueo pesimista de escritura con lectura y bloqueo al mismo tiempo.
Ilustración extraída de [McDonald, C. 2009]

```
// read  
Part p = em.find(Part.class, pId);  
// lock and refresh before update  
em.refresh(p, PESSIMISTIC_WRITE);  
int pAmount = p.getAmount();  
p.setAmount(pAmount - uCount);
```

Ilustración 5: Bloqueo pesimista de escritura con lectura y bloqueo posterior. Ilustración
extraída de [McDonald, C. 2009]

2. PESSIMISTIC_READ

- Evita que otras transacciones obtengan bloqueos PESSIMISTIC_WRITE, impidiendo modificaciones o borrados en otras transacciones. Se corresponde con un bloqueo compartido. Los proveedores de persistencia suelen dar soporte a este nivel pasando a PESSIMISTIC_WRITE, por lo que no suele utilizarse. En la Ilustración 6, se muestra un ejemplo.

```
//Read  
Part p = em.find(Part.class, pId);  
// lock p for update  
em.lock(p, PESSIMISTIC_READ);  
int pAmount = p.getAmount();  
p.setAmount(pAmount - uCount);
```

Lock after read, risk
stale, could cause
OptimisticLock
Exception

Ilustración 6: Bloqueo pesimista de lectura. Ilustración extraída de [McDonald, C. 2009]

3. PESSIMISTIC_FORCE_INCREMENT

- Evita que otras transacciones obtengan bloqueos PESSIMISTIC_READ o PESSIMISTIC_WRITE. Además el nº de versión (optimista) se incrementa en el commit (incrementando la versión incluso aunque no haya sido modificada). Se utiliza poco al combinar ambos métodos (pesimista y optimista).



Sin embargo, debemos recordar, que muchas bases de datos (entre ellas Oracle) **utilizan en la actualidad sistemas MVCC**, permitiendo realmente leer datos bloqueados para lectura, a partir de versiones previas de esos datos, no obteniéndose realmente dichos bloqueos de lectura.

4.3 Obtención de bloqueos

Para obtener un bloqueo se puede realizar de muy distintas formas en JPA:

- A través del método `lock` sobre el `EntityManager`
 - Ej: `em.lock(person, LockModeType.OPTIMISTIC);`
- Con el método `EntityManager.find`, pasando el tipo de bloqueo como argumento:
 - Ej: `Person person = em.find(Person.class, personPK, LockModeType.PESSIMISTIC_WRITE);`
- A través de la invocación del refresco de la entidad con `EntityManager.refresh`:
 - Ej: `em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);`
- Invocando a `Query.setLockMode` o `TypedQuery.setLockMode` pasando el modo de bloqueo como argumento:
 - Ej: `q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);`
- Añadiendo un elemento `lockMode` a la anotación `@NamedQuery`:
 - Ej: `@NamedQuery(name="lockPersonQuery", query="SELECT p FROM Person p WHERE p.name LIKE :name", lockMode=PESSIMISTIC_READ)`

En la siguiente Tabla 1, podemos ver como resumen final, el conjunto de bloques posibles, aunque tanto los bloqueos `READ`, `WRITE` y `NONE` están desuso (*deprecated*), en favor de los comentados previamente.

Lock Mode	Description
<code>OPTIMISTIC</code>	Obtain an optimistic read lock for all entities with a version attribute.
<code>OPTIMISTIC_FORCE_INCREMENT</code>	Obtain an optimistic read lock for all entities with a version attribute, and increment the version attribute value.
<code>PESSIMISTIC_READ</code>	Immediately obtain a long-term read lock on the data to prevent the data from being modified or deleted. Other transactions may read the data while the lock is maintained, but may not modify or delete the data.
<code>PESSIMISTIC_WRITE</code>	The persistence provider is permitted to obtain a database write lock when a read lock was requested, but not vice versa.
<code>PESSIMISTIC_FORCE_INCREMENT</code>	Immediately obtain a long-term write lock on the data to prevent the data from being read, modified, or deleted.
<code>READ</code>	A synonym for <code>OPTIMISTIC</code> . Use of <code>LockModeType.OPTIMISTIC</code> is to be preferred for new applications.
<code>WRITE</code>	A synonym for <code>OPTIMISTIC_FORCE_INCREMENT</code> . Use of <code>LockModeType.OPTIMISTIC_FORCE_INCREMENT</code> is to be preferred for new applications.
<code>NONE</code>	No additional locking will occur on the data in the database.

Tabla 1: Tipos de bloqueo en JPA



5. Caché

Un concepto fundamental para un rápido acceso a datos es el uso de las cachés. En JPA este concepto cobra gran importancia, puesto que asumimos una cierta degradación de rendimiento en este tipo de aplicaciones (ponemos **más capas** entre la base de datos y la aplicación). Para ello, se establecen distintos niveles de caché.

En la Ilustración 7, se muestra el esquema general de uso de cachés en JPA.

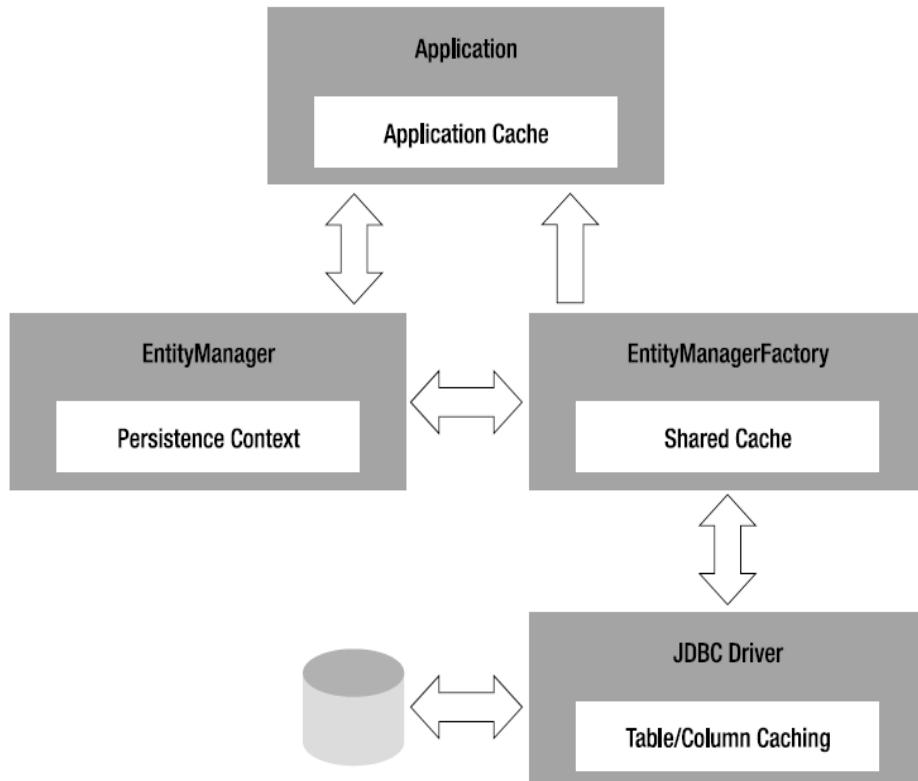


Ilustración 7: Cachés en JPA

Existe una caché de aplicación que es gestionada manualmente (se desaconseja su uso). Por encima de ésta, y actuando como intermediarias con el *driver JDBC* (con su propia caché de datos), tenemos dos niveles⁵:

- **Caché de nivel 1 (Level 1 / L1)** o caché del **contexto de persistencia**.
 - Garantiza una única entidad por fila de la base de datos.
- **Caché de nivel 2 (Level 2 / L2)** o **caché compartida**.
 - Comparte el estado entre entidades de distintos contextos.

5 Pueden existir más niveles de caché dependiendo del proveedor concreto, pero en JPA sólo se consideran estos dos niveles.



5.1 Caché del contexto de persistencia - Nivel 1

Está vinculada al `EntityManager` (sesión actual) y es utilizada en las búsquedas con `find` (a través del `EntityManager`) y en el recorrido de relaciones. Pero no se utiliza con consultas JPQL (ni Criteria API o nativas).

Las entidades gestionadas por el contexto de persistencia no se refrescan desde la base de datos, hasta que no se fuerza con una invocación al método `refresh`. En sentido inverso, no se sincronizan, llevando los datos a la base de datos, hasta que se realiza un `flush`.

Las entidades permanecen gestionadas en tanto en cuanto no se invoca al método `detach`, se realiza a una limpieza del gestor de entidades (e.g. `EntityManager.clear()`) o bien la transacción finaliza (ya sea con `commit` o `rollback`).

Se debe ser conscientes de que dichas entidades consumen recursos de memoria al estar en la caché. Recordemos que podemos consultar con el método `contains` si una entidad está gestionada por el `EntityManager` y por lo tanto en la caché de nivel 1.

Aunque en el caso de contextos gestionados por la aplicación, después de un `commit` podríamos seguir teniendo las entidades gestionadas. Finalmente, indicar que las cachés de nivel 1 son independientes entre sí, no compartiendo datos de caché entre distintas sesiones.

5.2 Caché compartida (Shared Cache) – Nivel 2

La caché compartida no se suele gestionar manualmente por parte del programador (como ocurre con la caché del contexto de persistencia a través de los métodos `detach` o `clear` del `EntityManager`) puesto que se trata de algo muy particular del proveedor de persistencia particular que se esté utilizando (quizás sólo justificable si estamos realizando *testing* de la aplicación).

En el Código 3, se muestra un ejemplo de invocación explícita de vaciado de caché compartida, a través de la interfaz `Cache`, en el código de un test con el framework JUnit. El objeto `Cache` se puede extraer del `EntityManagerFactory`, a través de su método `getCache`. La interfaz ofrece métodos:

- `contains` para consultar si un objeto está en caché.
- `evict` para desalojar un objeto o todos los objetos de un cierto tipo.
- `evictAll` para desalojar todos los objetos de la caché.



```
public class SimpleTest {
    static EntityManagerFactory emf;
    EntityManager em;
    @BeforeClass
    public static void classSetUp() {
        emf = Persistence.createEntityManagerFactory("HR");
    }

    @AfterClass public static void classCleanUp() {
        emf.close();
    }

    @Before public void setUp() {
        em = emf.createEntityManager();
    }

    @After public void cleanUp() {
        em.close();
        emf.getCache().evictAll(); // evict cache
    }

    @Test public void testMethod() {
        // Test code ...
    }
}
```

Código 3: Ejemplo de vaciado de caché compartida en test JUnit

La caché compartida (*Shared Cache* o L2) se puede configurar declarativamente⁶ en el fichero persistence.xml. Por ejemplo, con <shared-cache-mode>ALL</shared-cache-mode>.

Los valores que se pueden indicar son:

- ALL: todas las entidades van a caché compartida.
- NONE: desactiva la caché compartida.
- ENABLE_SELECTIVE: solo se activa para las entidades con @Cacheable(true).
- DISABLE_SELECTIVE: solo para aquellas entidades que no tienen @Cacheable(false).
- UNSPECIFIED: aplica el comportamiento por defecto del proveedor (valor por defecto⁷).

Los objetos POJO se pueden marcar para ser “cacheados” selectivamente, con la anotación @Cacheable, que recibe un argumento booleano, por defecto a true (es equivalente @Cacheable a @Cacheable(true)).

También se puede realizar una gestión dinámica de la caché, bien recuperando datos o en almacenamiento de datos con consultas, pasando la correspondiente propiedad al EntityManager, a la consulta directamente o estableciéndose como pista en la consulta posteriormente, con el método setHint.

En concreto proporciona las dos siguientes funcionalidades (si está habilitada):

- *On retrieval*: se utiliza para aquellos objetos que no están en el contexto de persistencia. Si no está disponible en la caché compartida, se recupera de la base de datos y se añade a la caché compartida. Cuando se utiliza find o queries.
- *On commit*: los objetos nuevos y modificados son añadidos a la caché compartida.

⁶ También se puede realizar vía código con la propiedad javax.persistence.sharedCache.mode, pasada al generar la fábrica de gestores de entidades.

⁷ Dicho valor se puede comprobar con PersistenceUnitInfo.getSharedCacheMode().



Para ajustar estos dos modos se pueden configurar los siguientes parámetros sobre el gestor de entidades (Ej: `entityManager.setProperty("javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS)`) o a nivel de una operación específica (Ej: `query.setHint(...)` o `find`):

- Propiedad `javax.persistence.cache.retrieveMode` en modo *on retrieval* con valores:
 - `CacheRetrieveMode.BYPASS`: saltar la caché e ir directamente a la base de datos.
 - `CacheRetrieveMode.USE`: usar primero la caché al leer entidades (valor por defecto).
- Propiedad `javax.persistence.cache.storeMode` en modo *on commit*, con valores nuevos o modificados:
 - `CacheStoreMode.BYPASS`: saltar la caché.
 - `CacheStoreMode.USE`: sitúa los objetos nuevos en caché cuando se obtienen de la BD o se hace `commit`, pero solo para aquellos objetos que no estaban en caché (valor por defecto).
 - `CacheStoreMode.REFRESH`: los nuevos datos son almacenados en caché, refrescando además los objetos entidad que ya estaban en la caché (útil si el valor ha cambiado en la base de datos externamente a JPA y se está usando `BYPASS` en *on retrieval*).

6. Resumen

En este tema se ha revisado brevemente el concepto de transacción en JPA, los bloqueos en accesos concurrentes y la gestión de caché en JPA. Son todos ellos conceptos generales, que envuelven al trabajo habitual con las entidades persistentes, y es necesaria una correcta configuración de estas facetas para completar un adecuado trabajo con JPA.

7. Glosario

Shared Cache: caché compartida.

8. Bibliografía

[Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol. ***Pro JPA 2. A definitive guide to mastering the Java Persistence API (2013)*** Apress. 2nd edition.

[Keith et al., 2018] Mike Keith, Merrick Schincariol, Massimo Nardone. ***Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence APIs.*** (2018) Apress. 3rd edition.

[Oracle, 2017] The Java EE 8 Tutorial (2017). Part VIII. Persistence. Disponible en <https://javaee.github.io/tutorial/toc.html>

[Oracle, 2014] The Java EE 7 Tutorial (2014). Part VIII. Persistence. Disponible en <http://docs.oracle.com/javaee/7/tutorial/>

McDonald, C. (2009) JPA 2.0. Concurrency and Locking. Oracle. Disponible en <https://dzone.com/articles/jpa-20-concurrency-and-locking>

Marques, G. (2015) JPA entity versioning (@Version and Optimistic Locking)



Aplicaciones de Bases de Datos

Grado en Ingeniería Informática

<http://www.byteslounge.com/tutorials/jpa-entity-versioning-version-and-optimistic-locking>

Marques, G. (2015) Locking in JPA (LockModeType)

<http://www.byteslounge.com/tutorials/locking-in-jpa-lockmodetype>

9. Recursos

Bibliografía complementaria:

Java Persistence. (2021, September 30). *Wikibooks, The Free Textbook Project*. Retrieved 08:21, April 9, 2022 from https://en.wikibooks.org/w/index.php?title=Java_Persistence&oldid=3992576



Licencia

Autores: Raúl Marticorena & Mario Martínez & Pablo García

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2022



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <https://creativecommons.org/licenses/by-nc-sa/4.0/>

