Curso JPA

Formación Sopra Steria
Formación JPA

Bloque JPA

Versión 1.0 del martes, 14 de diciembre de 2018

Historial

Versión	Fecha	Origen de la actualización	Redactado por	Validado por
1.0	14/12/2018		Alba Bermejo Solís Adrián Colmena Mateos Emilio Guillem Simón Nicole Tarela Duque Adrián Verdú Correcher Alejandro Mus Mejías	

Índice

1. Introducción

A nivel corporativo, el intercambio de información entre base de datos y aplicaciones funcionales se ha convertido en una de las necesidades básicas en cualquier sector comercial. El mercado de almacén de datos se ha visto incrementado a partir de la aparición de aplicaciones basadas en la nube. Sin embargo, aún se siguen desarrollando las tecnologías para el manejo eficiente de datos.

A pesar del éxito adquirido por la plataforma Java en el desarrollo con sistemas de bases de datos, durante un largo tiempo sufrió el mismo problema que muchos otros lenguajes de programación orientados a objetos. El intercambio de información entre base de datos y una aplicación Java era innecesariamente complicado. Los desarrolladores de Java se encontraban con la necesidad de escribir una gran cantidad de código para convertir las columnas y filas de la base de datos en objetos, o se hallaban sujetos a las especificaciones de los proveedores de *frameworks* que intentaban abstraer la complejidad de la base de datos. Afortunadamente, se introdujo en la plataforma un estándar como solución para eliminar la brecha existente entre los modelos de dominio orientado a objetos y los sistemas de base de datos relacionales: la API de Persistencia de Java (JPA).

Este manual introduce la versión 2.2 de la API de Persistencia de Java dentro del contexto de Java EE 8 y explora todo lo que ofrece a los desarrolladores.

Uno de los puntos fuertes de JPA es su portabilidad, la posibilidad de utilizarlo en distintos entornos (web, escritorio, etc).

En este primer capítulo se introducirá la motivación que justificó la aparición de JPA, así como una aproximación inicial a esta y sus conceptos fundamentales.

1.1. Bases de Datos Relacionales

Las bases de datos relacionales continúan siendo las más utilizadas a nivel comercial. En ellas se almacena la gran mayoría de los datos corporativos del mundo y siguen siendo el punto de partida de cualquier aplicación empresarial, donde se siguen utilizando actualización tras actualización.

Entender las bases de datos relacionales es fundamental para desarrollar aplicaciones empresariales y suele ser una de las partes que más tiempo conlleva dedicar. Uno de los principales motivos del éxito de Java, puede atribuirse a su fácil adaptación para crear sistemas de bases de datos corporativos.

1.2. Mapeo Objeto-Relacional

El modelo de dominio se define como el entorno de desarrollo donde se programe orientado a objetos. En este se habla de clases. Como en Java las clases definen objetos al instanciarse, a la hora de hablar del modelo de dominio también se llamará modelo de objetos.

En bases de datos se habla de tablas. Puede parecer que ambos modelos se asemejan lo suficiente como para tener una manera sencilla de conseguir relacionarlos pero las diferencias son importantes y la relación no es directa.

La técnica que consiste en relacionar el modelo de objetos y el modelo relacional es conocida como Mapeo Objeto-Relacional, a veces referido como mapeo O-R o simplemente ORM. Se llama mapeo

porque consiste en relacionar conceptos de un modelo con conceptos del otro, por ejemplo relacionar objetos a tablas.

Tres ideas claves en un ORM son:

- Debe poder programarse sin que la estructura de base de datos limite todo el desarrollo, pudiendo tener libertad de fijarse en el modelo de dominio más que en el modelo relacional.
- La aplicación Java siempre necesita tener control sobre los objetos que persiste y estar al tanto de su ciclo de vida.
- Es mucho más común que una aplicación se adapte a un esquema de base de datos (conjunto de tablas y sus relaciones) ya existente que crear uno nuevo desde cero. El soporte para esquemas heredados es uno de los casos más relevantes que pueden surgir.

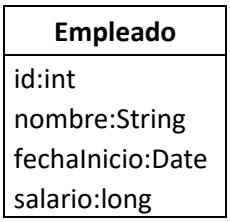
1.2.1. Impedance mismatch

A la diferencia entre el modelo objeto y el modelo relacional se le conoce comúnmente como *impedance mismatch*. El reto de mapear uno con el otro reside, no en las similitudes entre ambos, sino en los conceptos de ambos modelos que no tienen un equivalente lógico en el otro.

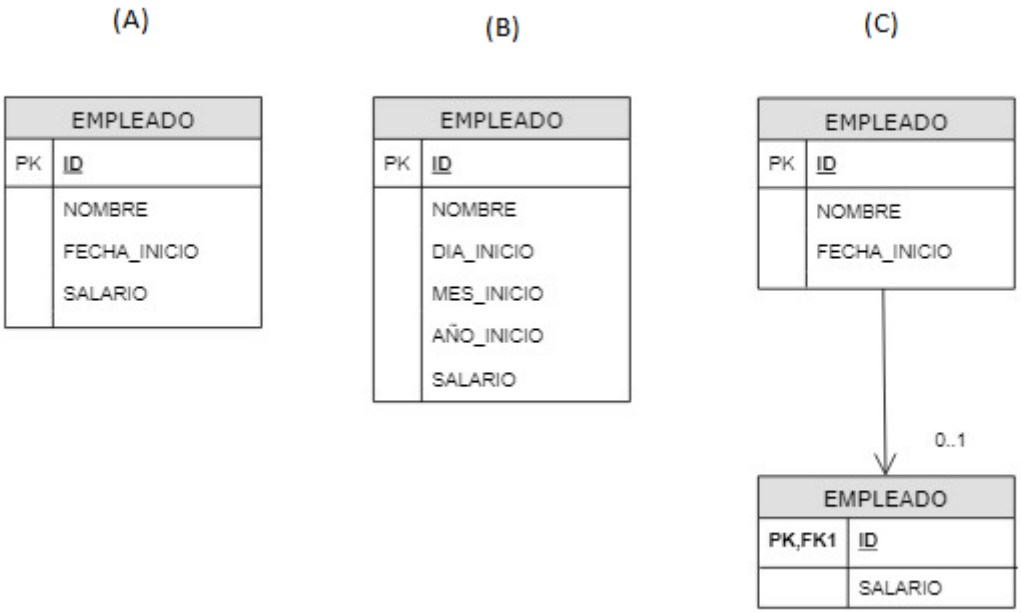
En las siguientes secciones se utilizan algunos modelos de dominio y la variedad de modelos relacionales que sirven para definir esos mismos modelos de dominio de distintas maneras . Como se verá, el reto en el mapeo objeto-relacional reside en la multitud de posibilidades existentes para un mismo caso.

1.2.2. Representación de Clases

Se empezará la discusión con una clase simple. La siguiente figura muestra la clase Empleado, la cual tiene cuatro atributos: id, nombre, fecha de inicio y salario.



En la siguiente figura, se tienen tres modelos relacionales distintos. La representación ideal de esta clase en la base de datos corresponde al caso (A). Cada atributo en la clase conecta directamente con una columna de la tabla. El ID del empleado es la primary key. Con la excepción de alguna variación en los nombres de los atributos, este caso es un mapeo directo.



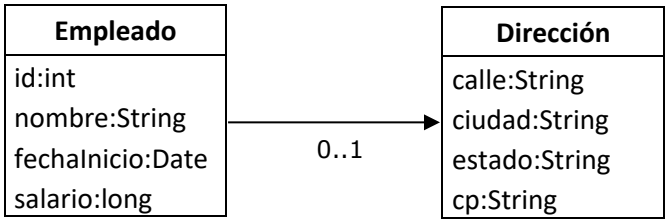
En el modelo (B), la fecha ha sido dividida en tres columnas distintas, DIA_INICIO, MES_INICIO y AÑO_INICIO, lo cual dificulta mucho el mapeo del atributo fecha al tener un formato distinto.

En este caso, el modelo (C) sería el más apropiado para la representación relacional de la clase Empleado, ya que el salario constituye una información sensible que, normalmente, se trata de manera separada al empleado en cuestión. Como se puede observar, el campo salario se almacena en una tabla separada, lo que permite restringir el acceso.

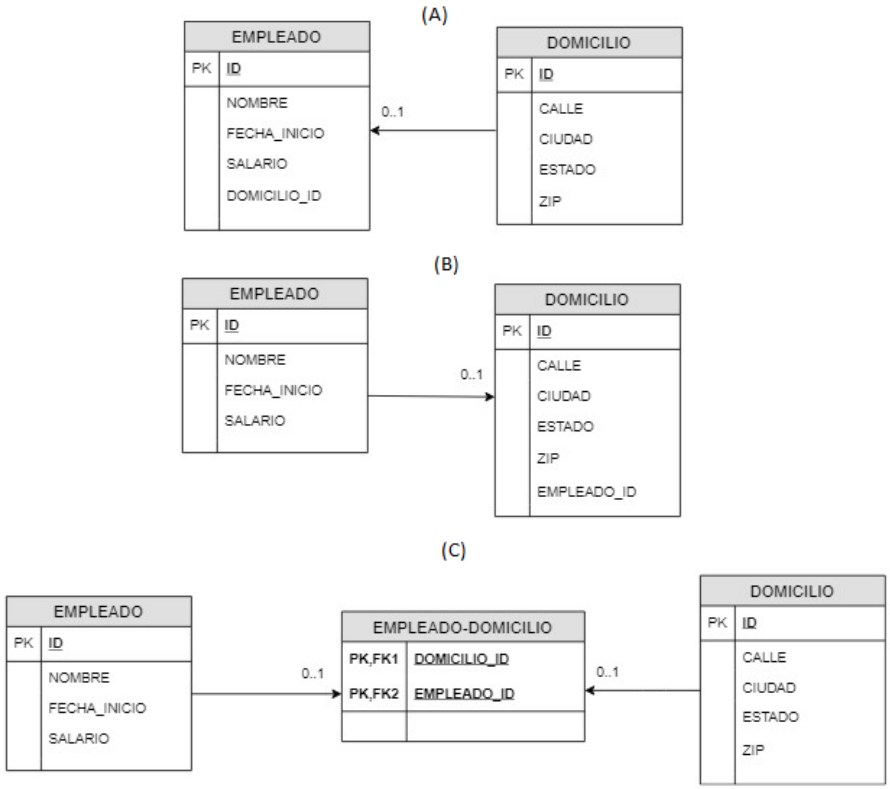
Se ha visto que incluso el mapeo de una única clase puede ser complicado, pero se debe recordar siempre que las necesidades de la base de datos priman ante las de la aplicación.

1.2.3. Relaciones

Las clases rara vez existen solas, lo normal es que están asociada con otras. Por ejemplo, a la clase Empleado, se le puede asociar una nueva clase, Domicilio, que de normal cada Empleado tendrá un Domicilio asociado, no más. En este caso se dice que Empleado y Domicilio tienen una relación one-to-one (uno a uno), que se representa en un diagrama UML (Unified Modeling Language) con 0..1. A continuación, se ilustra esta relación.



Al igual que en la sección anterior, en este caso también se tienen varias posibilidades para el esquema de base de datos, como se muestra en la figura siguiente.



El elemento esencial en torno al cual se construye la relación es la PRIMARY KEY. Para relacionar ambas tablas es necesario que ambas tengan una PRIMARY KEY , por lo que se le añade un id a Domicilio que actuará como PRIMARY KEY y, por tanto, se deberá adaptar el mapeo de alguna manera.

En el esquema (A) se muestra el mapeo ideal para la relación, introduciendo en la tabla Empleado la PRIMARY KEY de Domicilio, que actúa como FOREIGN KEY , de manera que se puede especificar el id del Domicilio al introducir un registro en la tabla Empleado.

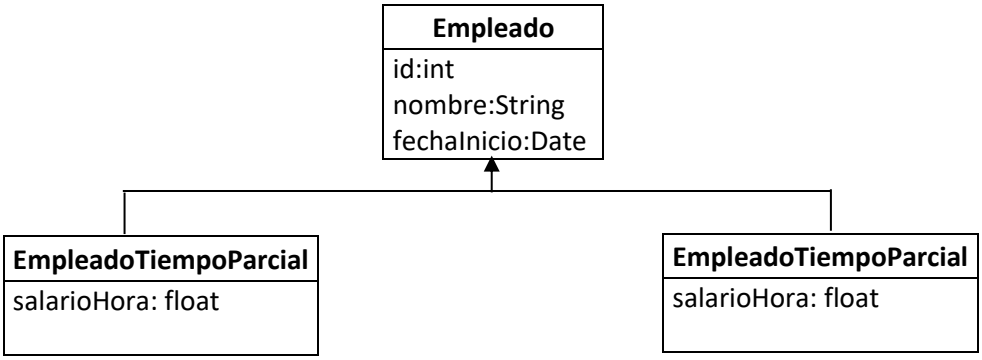
En cuanto al esquema (B), aunque muy similar al (A), es mucho más complejo, ya que en el modelo de dominio una instancia de Domicilio no tiene una referencia a una instancia de Empleado. El mapeo objeto-relacional debe tener en cuenta este desajuste entre la clase de dominio y la tabla o se deberá añadir una referencia al empleado para cada domicilio.

Para complicar más las cosas, el esquema (C) introduce una tabla de unión entre Empleado y Domicilio en la que se almacenan las combinaciones (Empleado/Direccion) de FOREIGN KEY de cada tabla. Con este diseño en cada operación que se haga contra la base de datos que involucre a ambas tablas se debe de ir a buscar la información en esa tabla de unión. Se podría modificar el modelo de dominio incluyendo una clase de asociación Empleado-Domicilio pero no es la solución intuitiva para modelar la relación entre la clase Empleado y la clase dirección.

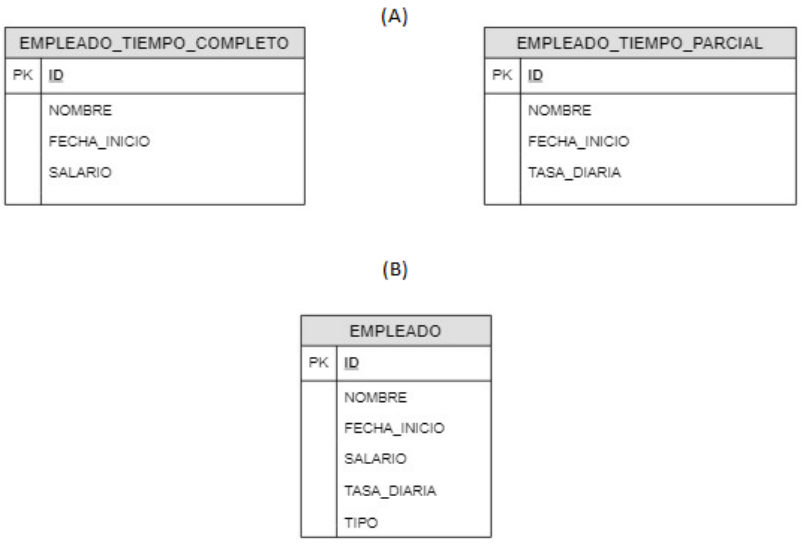
1.2.4. Herencia

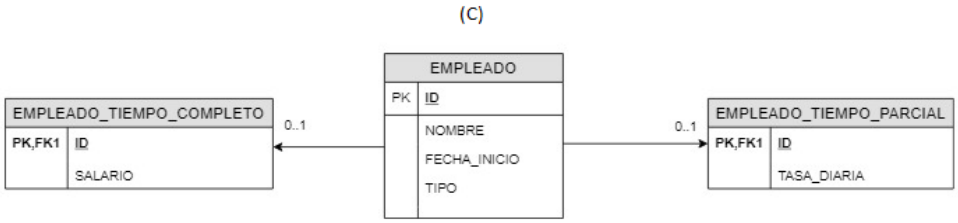
Lo que caracteriza al modelo de dominio es poder relacionar clases de manera que se cree una estructura jerárquica. Esto se denomina herencia y permite introducir el concepto de polimorfismo en una aplicación.

Retomando el ejemplo visto anteriormente de la clase Empleado, se puede pensar en una empresa que necesite distinguir entre empleados a tiempo completo y a tiempo parcial, que difieren únicamente en el tipo de salario. Esto supone una buena oportunidad para ilustrar el concepto de herencia. En java, para trabajar con objetos de tipo empleado, se crearía una clase Empleado que almacenaría los atributos comunes y dos subclases (EmpleadoTiempoCompleto y EmpleadoTiempoParcial) que almacenarían los atributos específicos de cada una, es decir el salario o la tasa diaria, como se puede ver en la figura siguiente. En estas subclases, además, se definirían las operaciones específicas de esa clase, como podría ser cambiar el salario.



En la figura que se verá a continuación, se muestran de nuevo tres posibles esquemas para la persistencia en base de datos.





Podría decirse que la solución más sencilla para el mapeo de una estructura de herencia en base de datos sería poner todos los campos necesarios para cada clase, incluyendo la clase madre Empleado, en tablas separadas, como ilustra el esquema (A). En este caso las tablas no tienen ninguna relación entre ellas, lo que supone que las consultas que impliquen a ambas serán mucho más complejas para el usuario.

Una solución más eficiente, aunque desnormalizada, se ilustra en el esquema (B), en el que todos los campos necesarios para cada clase se sitúan en una única tabla, lo que simplificaría enormemente las consultas. En este caso se introduce un nuevo campo TIPO que indica si el empleado es a tiempo parcial o completo, lo cual implicaría cambiar el modelo de dominio para interpretar la información aportada por cada registro en la tabla.

El esquema (C) lleva esto un paso más allá y normaliza los campos en tablas separadas para cada tipo de empleado. Al contrario que en el esquema (A), estas tablas están relacionadas por una tabla EMP que reúne los campos comunes a ambas. Puede parecer algo excesivo introducir una tabla nueva por una única columna extra, pero este método simplifica las consultas a realizar con ambas tablas y presenta los datos de manera lógica.

La herencia se convierte rápidamente en un tema complejo en el mapeo objeto-relacional. No sólo existe un reto con el almacenamiento de los datos de clase, sino que las relaciones entre tablas también dificultan las consultas eficientes.

1.3. **Resumen**

JPA es simple y elegante, potente y flexible. Su uso es natural y fácil de aprender. Cualquier API operativa podrá consistir de un pequeño número de clases.

1.3.1. Persistencia de los POJOs

Quizás lo más importante de JPA es que los objetos son POJOs (clases simples y con alta portabilidad), lo que significa que no hay nada especial en ningún objeto que se haga persistente. De hecho, casi cualquier objeto de aplicación no final existente, con un constructor predeterminado, puede hacerse persistente sin siquiera cambiar una sola línea de código.

1.3.2. No intrusivo

JPA existe como una capa separada de los objetos persistentes. Los objetos a persistir (Entity Beans) no necesitan implementar interfaces EJB (Entity Java Beans).

1.3.3. Object queries

JPA incluye QL (Query Language), un API basado en el lenguaje SQL que permite hacer consultas. De esta manera se pueden hacer consultas en la base de datos sin necesidad de conocer como están definidas físicamente las tablas y relaciones ya que los parámetros utilizados para construir las consultas son las clases y sus atributos Java. Estas consultas pueden devolver entidades (POJOS) pero también nuevos objetos, cálculos, etc

1.3.4. Mobile Entities

Las aplicaciones web y cliente/servidor son las más populares y estas aplicaciones suelen ser distribuidas, es decir se ejecutan en varias máquinas a la vez. JPA permite que las entidades sean móviles en la red, y que los objetos puedan "moverse" de una máquina a otra.

Cuando un objeto sale de la capa de persistencia se lo denominan *detached*. Una característica clave del modelo de persistencia de JPA es la capacidad de cambiar entidades *detached* y luego volver a unirlas (*attach*) cuando regresen a la máquina de origen.

1.3.5. Configuración simple

JPA tiene muchas opciones de configuración y todas ellas son configurables a través del uso de anotaciones, XML o una combinación de ambos. Las anotaciones Java (integradas en el código) permiten que el código sea fácil de usar y leer, y hacen posible que los principiantes pongan en marcha una aplicación rápida y fácilmente. Aun así, también se puede configurar JPA con ficheros XML como se ha hecho tradicionalmente en ORMs como Hibernate.

1.3.6. Integración y Tests

Normalmente las aplicaciones se ejecutan en Servidores de aplicaciones (contenedores), lo que complica hacer tests unitarios.

JPA está pensado para trabajar con contenedores, ya que es la forma habitual de desarrollo, pero a la vez permite que las aplicaciones desarrolladas con pocos cambios puedan ejecutarse relativamente fácil fuera del contenedor y también desarrollar tests unitarios que se ejecuten sin el contenedor.

1.4. **Conclusiones**

En este capítulo se ha presentado una introducción a JPA. Para ello, primero se inicia con una explicación de la necesidad de conectar Java con Bases de datos, y los problemas a los que se enfrentaron los desarrolladores, ilustrado con una serie de ejemplos y posibilidades para cada uno de los casos.

A continuación, se introduce el concepto de ORM, así como el tratamiento de este, y la necesidad de un estándar que fuera a la vez versátil y completo.

El capítulo se concluye con una breve revisión a la APP que incluye la historia de la misma y los proveedores que se unieron para crearla, introduciéndola en el marco de desarrollo de aplicaciones empresariales y, finalmente, describiendo algunas de las funcionalidades que ofrece la especificación.

2. Introducción a Entidades y Persistencia

El principal objetivo de JPA es que fuera fácil de usar y de entender, permitiendo a los desarrolladores definir y usar entidades de una forma sencilla e intuitiva.

En este capítulo se verán las características básicas de las entidades y los requisitos que deben tener. Además, se definirá el concepto de entidad y se explicará cómo crear, leer, actualizar y borrar una (Métodos CRUD). Se presentarán los objetos *EntityManager* y *Query* para poder ejecutar consultas a base de datos. Finalmente, se aplicarán todos los conceptos y métodos vistos sobre un ejemplo de aplicación.

2.1. Descripción general de una entidad

En general, una entidad es una representación en Java de una tabla de la base de datos que tiene una serie de características que la definen.

Y una entidad también se define como una clase en Java que tiene atributos y relaciones con otras entidades, así como la posibilidad de persistir dichos atributos y relaciones en una base de datos relacional.

En JPA cualquier clase definida en la aplicación puede ser una entidad. A continuación, se van a ver algunas de las características necesarias para que la clase se defina como una entidad:

1-Las entidades se persisten. Prácticamente cualquier clase puede persistirse en una base de datos así que puede ser una entidad. Para marcarla como entidad hay que declararla en la configuración de JPA de la aplicación en una *persistence unit*. Para ello se pondrá la anotación correspondiente en aquellas clases que sean entidades y estas se listarán en el fichero de configuración de persistencia XML.

2- Las entidades tienen que tener un identificador de persistencia (similar a una PRIMARY KEY en base de datos). Es un atributo o varios de la clase que sirven para identificar el campo (o los campos) de la base de datos con los que se corresponde la entidad.

3-Las entidades tienen un estado. En cualquier momento se tiene que poder saber si ha sido persistida (guardada en base de datos) o no.

4- Las entidades pueden ser tan simples como una clase con un solo atributo o tan complejas como una que tiene miles, pero para que se puedan usar lo ideal es que sean ligeras como un objeto normal de Java (no excedan su número de atributos).

2.2. Metadatos de las entidades

Además del estado de persistencia de la entidad (si los datos han sido guardados o no) cada entidad de la aplicación tiene metadatos asociados que la describen. Estos metadatos permiten que la capa de persistencia reconozca, interprete y administre correctamente la entidad desde el momento en el que se almacena hasta el momento de su uso en tiempo de ejecución.

Los metadatos necesarios para la creación de las entidades suelen ser mínimos y se pueden especificar mediante anotaciones o ficheros XML.

2.2.1. Anotaciones

Una anotación es una forma de añadir metadatos al código fuente que estén disponibles para la aplicación en tiempo de ejecución. JPA nos proporciona estas anotaciones, las cuales nos permiten definir entidades de una forma sencilla, al igual que nos permiten también definir el comportamiento de las diferentes propiedades de las clases donde se hayan.
Las anotaciones son la forma más simple de configurar JPA y tienen la ventaja de que se evita tener ficheros de configuración y que la entidad y sus metadatos están en el mismo fichero, es decir, en la clase Java, con lo que es muy rápido encontrar y consultar los metadatos de una entidad.
Con el uso de anotaciones, se puede convertir fácilmente una clase normal de Java en una entidad sin la necesidad de utilizar fichero de configuración XML. Todas las anotaciones que se van a ver a lo largo de este manual están definidas en el paquete `javax.persistence`.
La ventaja de usar anotaciones es que pueden ser usadas en cualquier clase, y permiten hacer el código de Java mucho más fácil de leer.

2.2.2. XML

Si quiere usarse XML, las anotaciones pueden ser cambiadas por descriptores de XML, ya que estos han sido modelados en su mayor parte a partir de las anotaciones.

2.2.3. Configuración por excepción (Default)

La configuración por excepción es otra forma de configurar JPA que hace que el motor de persistencia defina valores predeterminados, los que se aplican a la mayoría de las aplicaciones, y que el programador solo proporcione configuración adicional cuando desee sobrescribir algún valor. Estos valores predeterminados permiten que los metadatos sean más relevantes y concisos, ya que la configuración es más simple y solo lo que se sale de lo habitual. Sin embargo, cuando una aplicación utiliza configuración por excepción, también puede complicar el desarrollo ya que la configuración de JPA no es

visible. Esto puede hacer que los usuarios no sean conscientes de la complejidad de la configuración de persistencia.

Los valores predeterminados de configuración, permiten que un desarrollador se inicie de manera fácil y rápida con JPA, y luego simplemente deberá añadir nuevas configuraciones/metadatos a medida que aumente la complejidad de su aplicación y haya entidades o relaciones que salgan de lo habitual.

2.3. Creando una entidad

Las clases normales de Java son fácilmente transformables en entidades por medio de anotaciones, ya que cualquier clase con anotaciones y un constructor sin argumentos puede convertirse en una entidad.

Los requisitos mínimos que debe cumplir una entidad son los siguientes:

Debe tener importado el paquete `javax.persistence.Entity`.

- Debe tener como mínimo un constructor `public` o `protected` sin argumentos.
- No debe tener ningún método o variables declaradas como final.
- Debe implementar la interfaz `Serializable`.
- Puede heredar tanto de clases que ya son entidades como de clases de no sean entidad.
- Los atributos de las clases que son declaradas como entidad deben ser `private`, `protected` o `default` (es decir, solo pueden ser accedidos directamente por los métodos de la clase entidad).

Se procede a ver la creación de una clase normal de Java: `Empleado.class`, ilustrado en el Ejemplo 1.

Ejemplo 1. Clase Empleado

```
@Entity
public class Empleado {

 //definición de las propiedades
 @Id@ide
 private int id;
 @Column(name="nombre")
 private String nombre;
 @Column(name="salario")
 private long salario;

 //definición de los constructores(uno con argumentos y otro sin)
 public Empleado() {
 }

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

 //definicion de los métodos get y set de las propiedades
 public int getId() { return id; }
 public void setId(int id) { this.id = id; }
 public String getNombre() { return nombre; }
 public void setNombre(String nombre) { this.nombre = nombre; }
 public long getSalario() { return salario; }
 public void setSalario (long salario) { this.salario = salario; }
}
```

La anotación `@Id`, que sirve para identificar el atributo que es el identificador (equivalente a la clave primaria en la tabla de la base de datos), se coloca sobre la declaración de la propiedad privada o sobre su método *Getter*.

Los campos de las entidades se persisten automáticamente en la base de datos una vez esté persistida la entidad.

También es posible asignarle un nombre en concreto a la entidad añadiendo a la anotación `@Entity` un elemento `name`. Por ejemplo, `@Entity(name= "Emp")`, lo cual permite identificar a la entidad en la base de datos por el nombre que se le ha asignado.

En cambio, si no se añade ningún nombre a la entidad, la tabla tomará el nombre por defecto de la clase. En el ejemplo anterior, si no le se le pone el campo `name` a la anotación `@Entity`, el nombre por defecto de la entidad será "Empleado" y la tabla correspondiente a esta entidad en base de datos saldrá con el nombre "EMPLEADO".

De la misma manera sucede con los campos. Si no se especifica ningún nombre para las diferentes columnas de la tabla, las columnas cogerán el valor por defecto de los campos de la entidad. El nombre de los campos se puede cambiar añadiendo la anotación `@Column(name="nameColumn")`. Si la tabla ya estuviera creada, los campos `name` (tanto de esta anotación como de la anterior) tendrían que coincidir con los nombres de las tablas para que JPA pueda relacionarlas.

2.4. Entity Manager

Para persistir una entidad en la base de datos con JPA, se hace una llamada a su API. Esta API es implementada por el *Entity Manager* (EM) y encapsulada casi por completo en una única interfaz llamada `javax.persistence.EntityManager`. Hasta que un EM no crea, lee o escribe una entidad, esta no es más que un Objeto Java normal.

Cuando un EM es llamado por una entidad, este objeto pasará a estar gestionado por el propio EM. El grupo de entidades gestionadas dentro de un EM en un momento dado se llama contexto de persistencia. No pueden existir simultáneamente dos instancias de Java con la misma clave primaria en el mismo contexto de persistencia. Por ejemplo, no pueden existir dos empleados que tengan un ID con valor 12. El EM se encarga de hacer este tipo de verificaciones.

Existen distintos EM para trabajar con una base de datos u otra. Cada uno de ellos forma parte de un *persistence provider* diferente.

Es el *provider* el que suministra el motor de implementación de soporte para todo JPA, desde el *EntityManager* hasta la implementación de clases *Query* y la generación de SQL.

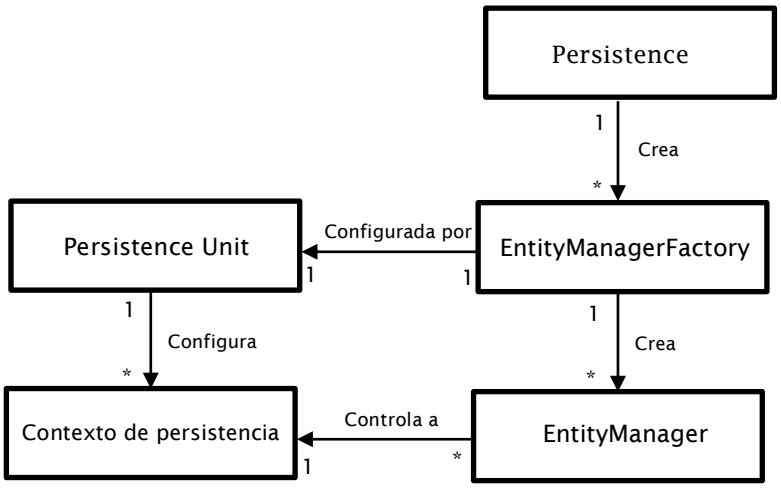
Todos los EM provienen de *factories* del tipo `javax.persistence.EntityManagerFactory`.

La ventaja de crear un método *Factory*, en este caso *EntityManagerFactory*, es que el programador pueda desarrollar distintos EMs, pudiendo guiarse todos por una misma configuración definida por este *EntityManagerFactory* que se relacionará directamente con la *persistence unit*, la cual establecerá los ajustes y las clases de entidad usadas por los EMs. Esto permite que el código pueda ser usado para cualquier persistence *provider*, es una capa intermedia que permite ordenar y estandarizar los distintos EMs que el programador cree.

Las *persistence units* se nombran para diferenciar una *EntityManagerFactory* de otra. Esto le da a la aplicación control sobre qué configuración o persistencia debe utilizarse para operar en una entidad en particular.

La siguiente figura muestra que por cada *persistence unit* hay una *EntityManagerFactory* y que muchos EMs pueden ser creados a partir de una *EntityManagerFactory*. No obstante, varios EMs pueden apuntar

al mismo contexto de persistencia. Sólo se ha hablado de un EM y su contexto de persistencia, pero más adelante se verá que puede haber múltiples referencias a diferentes EMs, todos apuntando al mismo grupo de entidades administradas.



Objeto	Objeto del API	Descripción
Persistence	*Persistence*	Clase de *bootstrap* que sirve para obtener un *EntityManagerFactory*
EntityManagerFactory	*EntityManagerFactory*	Objetos necesarios para obtener los *EntityManagers*
Persistence Unit	--	Configuración con nombre que declara las entidades y como almacenar los datos
EntityManager	*EntityManager*	API principal que sirve para realizar operaciones y *queries* en las entidades
Contexto de persistencia	--	Conjunto de todas las instancias de la entidad administradas por un EM específico

2.4.1. Obtener un Entity Manager

Los EM siempre se obtienen de un *EntityManagerFactory,* el cual determina la configuración de sus EMs. El método estático `createEntityManagerFactory()` en una clase devuelve el *EntityManagerFactory* para la *persitence unit* indicada. El próximo ejemplo indica cómo se crearía un *EntityManagerFactory* para la unidad de persistencia llamada ServicioEmpleado:

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("ServicioEmpleado");
```

La *persistence unit* ServicioEmpleado determina que la configuración sea la de ese *EntityManagerFactory*, especificando sus atributos tales como los parámetros de conexión que los EMs usarán para conectarse a la base de datos. Ahora obtener un EM será tarea fácil:

```
EntityManager em = emf.createEntityManager();
```

El motivo de crear distintos EM, es el poder controlar qué y cuándo se persiste algo en base de datos, ya que como el texto explica, por muchos cambios que se realicen en el código, en base de datos sólo se reflejarán cuando se realice el *commit*.

2.4.2. El método persist

Persistir una entidad es la operación de tomar una entidad transitoria, o una que todavía no tiene ninguna representación persistente en la base de datos, y almacenar su estado para que pueda ser recuperada más tarde. Se usa el EM para persistir en una instancia de `Empleado`, como en este ejemplo:

```
Empleado emp = new Empleado(158);
em.persist(emp)
```

La primera línea en este código crea una instancia de `Empleado` que se quiere persistir (solo fijando el ID, no el nombre o salario), la cual es sólo un objeto de Java. La segunda línea usa el EM para persistir la entidad. Si encuentra algún problema lanzará una `PersistenceException`.

El siguiente ejemplo muestra cómo crear un método simple que cree un nuevo empleado y lo persista en la base de datos.

Ejemplo 2. Método para crear un empleado.

```
public Empleado crearEmpleado(int id, String nombre, long salario) {
Empleado emp = new Empleado(id);
emp.setNombre(nombre);
emp.setSalario(salario);
em.persist(emp);
return emp;
}
```

Este método utiliza el *EntityManager* em para persistir `Empleado`. No es necesario preocuparse por un error de ejecución ya que en caso de existir uno se lanzará una *PersistenceException*.

2.4.3. El método find

Cuando una entidad se encuentre en base de datos, será habitual querer recuperarla. Con esta línea puede encontrarse:

```
Empleado emp = em.find(Empleado.class, 158);
```

Es necesario escribir la clase de la entidad que se esté buscando y el valor de su clave primaria. Una vez encontrado el empleado existirá en un contexto de persistencia actual asociado a ese EM. Al especificarse la clase, se evita la necesidad posterior de un *cast* de objeto (lo que devuelve el *find* por defecto) a la clase de nuestra entidad.

Si para buscar un objeto se utiliza un ID incorrecto (un objeto que haya sido borrado o que nunca haya existido) devolverá *null*, por lo que antes de utilizar emp debe verificarse si el valor de este es *null*.

A continuación, se muestra un sencillo ejemplo del método *find*.

Ejemplo 3. Método para encontrar un Empleado

```
public Empleado encontrarEmpleado(int id) {
   return em.find(Empleado.class, id);
}
```

2.4.4. El método remove

Si se quiere ejecutar lo que sería en SQL un DELETE en una o más tablas, el método *remove* cobra sentido.

Para poder eliminar una entidad, esta tiene que estar en el contexto de persistencia. Es decir, antes de eliminarla, la aplicación debe haber recuperado a esa entidad a través del EM, como puede verse a continuación:

```
Empleado emp = em.find(Empleado.class, 158);
em.remove(emp);
```

Primero, la entidad debe ser encontrada utilizando `find()`, que devolverá una instancia de Empleado, y luego eliminarla utilizando `remove()`. La excepción `java.lang.IllegalArgumentException` aparecerá si el valor de `find()` es *null*. Esto puede solucionarse comprobando que existe el empleado antes de intentar borrarlo, como en el Ejemplo 4.

Ejemplo 4. Método para eliminar un empleado.

```
public void removeEmpleado(int id) {
    Empleado emp = em.find(Empleado.class, id);
    if (emp != null) {
 em.remove(emp);
    }
}
```

2.4.5. Modificando una entidad

Hay varias maneras de modificar una entidad, pero a continuación se muestra una de las maneras más simples y utilizadas. Para ello, se debe encontrar una entidad y una vez hallada, cambiar alguno de sus campos. En las próximas líneas se muestra un ejemplo en el que se modifica el salario del empleado 158.

```
Empleado emp = em.find(Empleado.class, 158);
emp.setSalario(emp.getSalario() + 1000);
```

En el ejemplo se muestra cómo modificar el sueldo de un empleado, aunque este no se reflejará en la base datos hasta que se llame al método *commit*, en la transacción.

Ejemplo 5. Método para modificar un Empleado.

```
public Empleado subirSueldoEmpleado(int id, long raise) {
    Empleado emp = em.find(Empleado.class, id);
    if (emp != null) {
        emp.setSalary(emp.getSalary() + raise);
    }
    return emp;
}
```

El método devolverá el empleado ya modificado.

2.4.6. Transacciones

En los ejemplos anteriores no se han incluido las transacciones a pesar de que son necesarias si lo que se quiere es realizar un cambio persistente en la entidad. A excepción de *find(),* el cual puede ser llamado con o sin transacción debido a que no realiza ningún cambio en la base de datos, todos los demás métodos necesitan del uso de transacciones.

A la hora de realizar una transacción, se utiliza la API de transacción estándar de Java (JTA). Las transacciones, en caso de no existir, provocarán la aparición de una excepción y además los cambios no llegarán a persistirse en la base de datos.

En el entorno en el que se está trabajando en este manual, Java SE, se necesita iniciar y cerrar una transacción con el servicio `javax.persistence.EntityTransaction` a la hora de realizar cualquier modificación de una entidad. Es decir, se debe iniciar la transacción, realizar el cambio pertinente y cerrar la transacción. La transacción se llama mediante el método `getTransaction()` del EM para obtener un *EntityTransaction* y luego llamar a `begin()`. Del mismo modo que para cerrar una transacción se llama al método `commit()` sobre el objeto *EntityTransaction*. A continuación, se muestro un ejemplo de esto:

Ejemplo 6. Abriendo y cerrando una *EntityTransaction*

```
em.getTransaction().begin();
createEmpleado(158, "John Doe", 45000);
em.getTransaction().commit();
```

2.4.7. Queries

Una *query* de JPA se diferencia de utilizar JDBC para enviar una *query* SQL, en que permite recuperar objetos de tipo entidad en vez de un objeto genérico de resultado (*ResultSet*) que haya que convertir luego a mano en entidades.

En JPA, una *query* es similar a una consulta en base de datos, pero en lugar de utilizar SQL (*Structured Query Language*) se utiliza un lenguaje llamado JP QL (*Java Persistence Query Language*).

Para crear una consulta en JPA es necesario pasar por el EM y llamar a su método createQuery() pasándole parámetros para definir la consulta.

Hay dos tipos de consultas, estáticas o dinámicas. Una consulta estática se define normalmente mediante anotaciones o con XML *metadata*, y debe incluir el *query criteria* y un nombre asignado por el usuario. Este tipo de consultas son las llamadas *named queries*, y se buscan por su nombre en el momento de la ejecución del programa.

Por otro lado, una consulta dinámica se puede crear en el momento de la ejecución proporcionando el *query criteria* JP QL o un objeto de criterio. Pueden ser un poco más difíciles de ejecutar ya que el *persistence provider* no puede hacer ninguna preparación previa, pero las consultas JP QL son muy simples de utilizar y pueden responder ante la lógica del programa o incluso ante la lógica de usuario.

En la versión de JPA 2.2 se introduce un nuevo método en las consultas llamado `getResultStream()`, que devuelve como resultado de la consulta un *stream* de Java 8. Este nuevo método de crear consultas aprovecha la nueva funcionalidad de expresiones lambda de Java 8 y permite explotar más fácilmente el resultado ya que se devuelve en forma de *stream*.

En el siguiente ejemplo se muestra cómo crear una consulta dinámica sencilla y ejecutarla para obtener todos los empleados de la base de datos.

Ejemplo 7. Ejemplo de consulta dinámica.

- Usando JPA 2.1:

```
TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e", Empleado.class);
List<Empleado> emps = query.getResultList();
```

- Usando JPA 2.2

```
Stream<Empleado> Empleado = em.createQuery("SELECT a FROM Empleado e",
Empleado.class).getResultStream();
```

Se crea un objeto `TypedQuery<Empleado>` mediante la llamada `createQuery` en el *EntityManager* y pasando en la cadena JP QL que especifica el *query criteria*, así como la clase en la que la *query* debe estar parametrizada. La cadena JP QL se refiere a la entidad Empleado, no a la tabla Empleado de la base de datos, así que esta *query* selecciona todos los objetos Empleado sin filtrarlos.

Al igual que al usar JPA 2.2, el nuevo método `getResultStream()` devuelve una secuencia Java 8 del resultado de la *query*. Así que, en este caso, devolverá el *stream* del resultado de la *query* Empleado.

Estos ejemplos muestran lo sencillo que es de crear, ejecutar y procesar *queries*, pero no muestran lo poderosas que son.

2.5. Recopilación

Una vez vistos todos los métodos anteriores, la idea es juntarlos y combinarlos en una única clase. Esta actúa como una clase de servicio, que se llamará `ServicioEmpleado`, y que permitirá realizar operaciones con los empleados.

A continuación, muestra la implementación de todos los métodos en la clase mencionada.

Ejemplo 8. Clase de servicio para operar con la entidad Empleado.

```
import javax.persistence.*;
import java.util.List;

public class ServicioEmpleado {
 protected EntityManager em;

 public ServicioEmpleado (EntityManager em) {
        this.em = em;
 }
```

```
public Empleado crearEmpleado (int id, String nombre, long salario) {
        Empleado emp = new Empleado(id);
        emp.setNombre(nombre);
        emp.setSalario(salario);
        em.persist(emp);
        return emp;
}

public void borrarEmpleado(int id) {
        Empleado emp = buscarEmpleado(id);
        if (emp != null) {
                em.remove(emp);
        }
}


public Empleado aumentarSalario (int id, long aumento) {
          Empleado emp = em.find(Empleado.class, id);
          if (emp != null) {
               emp.setSalario(emp.getSalario() + aumento);
          }
          return emp;
}

public Empleado buscarEmpleado (int id) {
   return em.find(Empleado.class, id);
}

public List<Empleado> buscarTodosLosEmpleados () {
   TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e",
        Empleado.class);
   return query.getResultList();
}
}
```

Esta sencilla clase puede ser utilizada para realizar las funciones básicas tales como crear, leer, modificar y borrar (CRUD) en la entidad Empleado. La clase exige crear un *EntityManager* para iniciar (método `em.begin()`) y llevar a cabo (método `em.commit()`) las transacciones especificadas. De esta

manera, al establecer de manera independiente la lógica de transacción de la lógica de operación, el código está menos acoplado y es más reutilizable.

A continuación, se muestra un programa *main* muy sencillo que utiliza este servicio y lleva a cabo todas las acciones requeridas por el *EntityManager* y el *transaction managment*.

Ejemplo 9. Método *main* que utiliza la clase ServicioEmpleado.

```
import javax.persistence.*;
import java.util.List;

public class EmpleadoTest {

	public static void main(String[] args) {

		EntityManagerFactory emf =
		Persistence.createEntityManagerFactory("ServicioEmpleado");
		EntityManager em = emf.createEntityManager();
		ServicioEmpleado servicio = new ServicioEmpleado(em);

		// crear y persistir un empleado
		em.getTransaction().begin();
		Empleado emp = servicio.crearEmpleado(158, "John Doe", 45000);
		em.getTransaction().commit();
		System.out.println("Empleado persistido " + emp);

		// buscar un Empleado determinado
		emp = servicio.buscarEmpleado(158);
		System.out.println("Encontrado " + emp);

		// buscar todos los empleados
		List<Empleado> emps = servicio. buscarTodosLosEmpleados ();
		for (Empleado e : emps)
		System.out.println("Empleados encontrados: " + e);

		// modificar empleado
		em.getTransaction().begin();
		emp = servicio.aumentarSalario (158, 1000);
		em.getTransaction().commit();
		System.out.println("Modificado " + emp);
```

```
                // borrar un empleado
                em.getTransaction().begin();
                servicio.borrarEmpleado(158);
                em.getTransaction().commit();
                System.out.println("Empleado 158 borrado");

                // cerrar el EM y EMF
                em.close();
                emf.close();
        }
}
```

Como se puede ver, al final del programa tanto el *EntityManager* como el *EntityManagerFactory* son cerrados mediante el comando `close()`, lo cual es muy importante, ya que nos asegura la correcta realización de las acciones y que todos los cambios han sido registrados.

2.6. Uniendo todas las piezas

Una vez conocidas todas las piezas clave de JPA, es hora de introducirlas todas para realizar una aplicación con Java SE, que será lo que se llevará a cabo en esta última parte del capítulo.

2.6.1. Persistence Unit

La configuración que describe la *persistence unit* está definida en un archivo XML llamado *persistence.xml*. Este debe colocarse en una localización particular para que pueda ser encontrado por JPA, en la carpeta META-INF. Cada *persistence unit* tiene un nombre propio, de manera que cuando una aplicación quiere especificar la configuración, únicamente necesita hacer referencia al nombre de la *persistence unit* en la que se define dicha configuración. Un archivo *persistence.xml* puede contener una o varias configuraciones de *persistence unit*, pero cada una de ellas es distinta e independiente de las otras y no se encuentran relacionadas entre ellas. Esto podría ser útil en casos particulares, pero en general se tendrá una única *persistence unit*.

Para este caso, únicamente son necesarias tres secciones, llamadas *transaction-type*, *class* y *properties*. El Ejemplo 10 muestra las partes más relevantes del archivo *persistence.xml* para este caso.

Ejemplo 10. Elementos en el archivo *persistence.xml*.

```
<persistence>
<persistence-unit name="ServicioEmpleado " transaction-type="RESOURCE_LOCAL">
<class>examples.model.Empleado</class>
<properties>
<property                                          name="javax.persistence.jdbc.driver"
 value="org.apache.derby.jdbc.ClientDriver"/>
<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/EmpServDB;create=true"/>
<property name="javax.persistence.jdbc.user" value="APP"/>
<property name="javax.persistence.jdbc.password" value="APP"/>
   </properties>
</persistence-unit>
</persistence>
```

El atributo *name* de la *persistence unit* indica su nombre y es la línea que se especifica cuando se crea el *EntityManagerFactory*, en este caso "ServicioEmpleado". El atributo *transaction-type* sirve para configurar las transacciones y, en este caso, indica que la *persistence unit* usa el nivel de recursos del *EntityTransaction* en lugar de las transacciones de JTA. El elemento *class* enumera las clases que forman parte de la *persistence unit*. Cuando hay más de una entidad se pueden enumerar múltiples clases en esta lista. En Java EE, cuando usa un contenedor (*container)*, no suele ser necesario detallar las clases ya que el *container* buscará automáticamente las clases anotadas como `@Entity` como parte del proceso de despliegue de la aplicación, pero son necesarias para la ejecución simple en una aplicación de consola en Java SE. En ese ejemplo sólo existe una entidad Empleado.

La última sección consiste en una lista de propiedades, parámetros de configuración, que pueden ser estándar de JPA o específicas del distribuidor. En este caso son parámetros como el nombre de usuario y contraseña de la base de datos, datos que habitualmente se especifican en este archivo para que JPA sepa dónde debe realizar las acciones y la conexión se lleve a cabo con éxito.

3. ORM

Este capítulo se centrará en el ORM o **Mapeo de Objetos Relacional** (*Object-Relational Mapping*).

El ORM es el componente de un API encargado de persistir objetos en una base de datos relacional. La mayor parte del API está dedicada al ORM.

Los grandes bloques que se tratarán son:

1) Mapeo de una entidad a una base de datos. Cómo se relaciona una clase Java con una tabla de una base de datos relacional.
2) Mapeo de las relaciones entre las distintas entidades.
3) Objetos embebidos.

En cada sección se darán ejemplos para aclarar los conceptos y la sintaxis aprendida.

3.1. Contexto

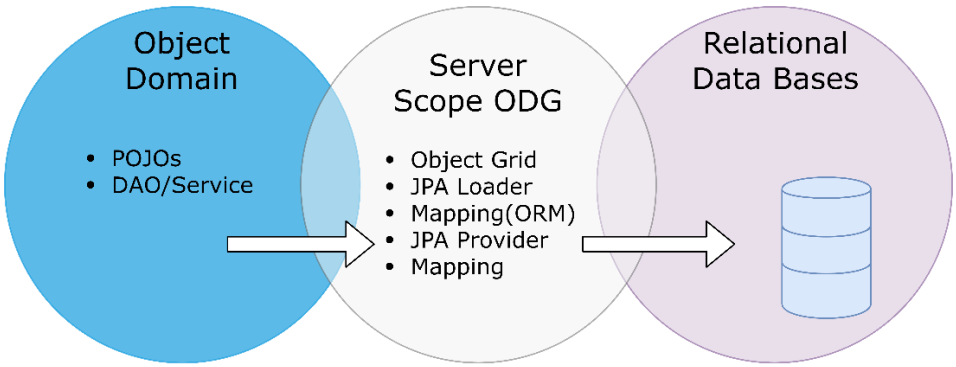
3.1.1. Alcance del ORM

Cuando se habla de JPA se hace mención a una poderosa herramienta que es capaz de comunicar dos mundos funcionalmente distintos. Esto es posible gracias al Mapeo de Objetos Relacional. El ORM suele encontrarse en todas las etapas a la hora de persistir un objeto, desde el mapeo de las clases Java hacia sus tablas equivalentes en la base de datos relacional, hasta cómo emitir consultas a través de los objetos.

En este capítulo se introducirán las nociones básicas del mapeo de campos de una misma entidad (*Entity Fields*), así como los diversos conceptos y métodos que describen a las relaciones entre diferentes entidades.

Las características más relevantes del ORM son:

- Persistencia Idiomática, la declaración de entidades se produce en el dominio de la Programación Orientada a Objetos.
- Alto rendimiento, enfocado a favorecer la concurrencia por medio del *fetching* y *locking*.
- Solidez, siendo un componente bien fundamentado que ha sido diseñado para perdurar.



3.1.2. Introducción a las Anotaciones

Antes de continuar con el capítulo y dado que se va a trabajar extensamente con anotaciones, es oportuno dedicar unas líneas a sus conceptos fundamentales.

Las anotaciones de persistencia aparecen en tres niveles distintos. Estos son:

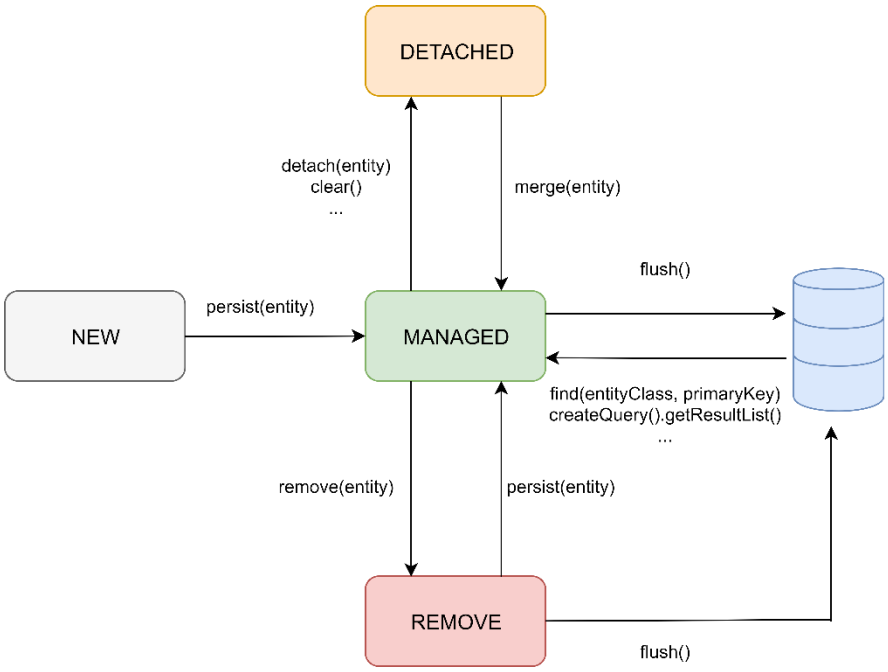
- Clase
- Atributo
- Método

Estas, pueden aparecer en la línea inmediatamente superior del código, así como en la misma línea, sin influir en el funcionamiento. La elección, en cada caso, persigue únicamente la obtención de un código claro y legible.

En JPA las anotaciones fueron diseñadas buscando la sencillez y flexibilidad. Por ello, es normal encontrar a menudo más de una anotación para una misma porción de código. Eso se puede dar como anotaciones hermanas (*sibling annotations* –una detrás de otra-) o como anotaciones anidadas (*nested anotations* -una dentro de otra-). *Sibling annotation* es la elección de diseño predominante. Pero como todas las elecciones, esta tiene un precio y es que permite permutaciones en los metadatos sintácticamente correctas, pero semánticamente inválidas. Ante este tipo de situaciones, aunque el compilador pase por alto las anotaciones, el proveedor de persistencia no. Es por ello que, al realizar la lectura, **ignora** todas las anotaciones que considera inaceptables en un mismo grupo. Este es un comportamiento relevante que puede pasar desapercibido y puede ser la causa de un funcionamiento inesperado a la hora de probar una aplicación.

3.2. Entity State

El *entity state* indica al proveedor de persistencia el punto en el que se encuentra el objeto a persistir. El ciclo entre los diferentes estados posibles, se puede ver en el siguiente diagrama.



El proveedor de persistencia tiene que tener acceso en todo momento al estado de la entidad (*entity state*), ya sea cuando se esté recuperando (en el contexto de persistencia) un dato de la base de datos (e.g. *find(entityClass, primaryKey)…*), o cuando se quiera persistir un objeto Java (e.g. *persist(entity)…*).

Independientemente del origen, la manera de acceder al estado de la entidad se conoce como *access mode*. Existen dos comportamientos asociados al *access mode* que dependen de cómo haya sido definido:

- Si se realizan las anotaciones **en los atributos** de la clase, el proveedor accederá a los datos de la entidad usando reflexión. Esto se conoce como ***Field Access***.
- Si las anotaciones se definen **en los métodos** *Getter,* el proveedor de persistencia llamará a los métodos *Getter* y *Setter* para gestionar el estado. Esto se conoce como ***Property Access***.

A continuación, se explicará detalladamente las diferencias entre los tipos de acceso.

3.2.1. Diferencias entre los tipos de Acceso

A la hora de utilizar *Field Access* es importante tener en cuenta que a pesar de estar declarados explícitamente los métodos *Getter* y *Setter*, estos serán **ignorados** por el proveedor de persistencia. Los atributos pueden ser declarados como `protected`, `default` o `private`, y nunca como `public` puesto que eso abriría la clase permitiendo que se sobrepasase la implementación del proveedor. Lo recomendado es obligar a cualquier otra clase a utilizar los métodos de una entidad para acceder a su estado de persistencia. De hecho, incluso la clase misma no debería manipular de manera directa a los campos de su entidad salvo durante la inicialización.

Ejemplo 1. Uso de *Field Access*

```
@Entity
public class Employee {
  @Id private long id;
  private String name;
  private long salary;

  public long getId() { return id; }
  public void setId(long id) { this.id = id; }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public long getSalary() { return salary; }
  public void setSalary(long salary) { this.salary = salary; }
}
```

El ejemplo anterior ilustra el uso de *Field Access* en el mapeo de una entidad. La anotación `@Id`, en este caso, indica que el atributo `id` de la clase `Employee` es el identificador de persistencia (PRIMARY KEY de

la tabla EMPLOYEES). Pero también, que el acceso al e*ntity state* será un *Field Access*, puesto que la anotación aparece encima de un atributo, asumiéndose el mapeo de los campos `name` y `salary` por defecto.

Por otro lado, si se emplea *Property Access*, los métodos *Getter* y *Setter* son lógicamente **obligatorios**. Bajo este paradigma, el tipo de dato que le corresponde a la entidad para el mapeo de campos se obtiene por medio de *returnType* del *Getter* que debe ser el mismo tipo de dato que se le pasa por parámetros al *Setter.* La declaración de estos métodos debe ser `public` o `protected`. Como se ha mencionado previamente, las anotaciones para el mapeo de una propiedad deben aparecer en el *Getter*.

Ejemplo 2. Uso de *Property Access*

```
@Entity
public class Employee {
 private long id;
 private String name;
 private long wage;

 @Id public long getId() { return id; }
 public void setId(long id) { this.id = id; }
 public String getName() { return name; }
 public void setName(String name) { this.name = name; }
 public long getSalary() { return wage; }
 public void setSalary(long salary) { this.wage = salary; }
}
```

En este ejemplo se puede apreciar el uso de *Property Access,* donde la anotación `@Id` que aparece en el `getId()` definirá el acceso al *entity state* para el proveedor de persistencia. Como aparece en este caso, el mapeo por defecto de las propiedades de la entidad se da a través de los nombres de método en los *Getter* y *Setter* que deberán coincidir con los nombres de las columnas de la tabla *EMPLOYEES*. Adicionalmente se ha cambiado el nombre de uno de los atributos de clase, `wage`, para ilustrar que con este tipo de acceso ya no es necesario aquí el uso de un nombre identificativo.

Antes de concluir cabe mencionar que hay otro tipo de acceso que no ha sido mencionado con anterioridad. Este, no es un tipo de acceso en sí, sino la combinación de los dos anteriores y es conocido como ***Mixed Access.***

Para este caso, la declaración del acceso debe hacerse de forma explícita por medio de la anotación `@Access`. Aunque no es muy común, puede resultar muy útil en algunos casos.

Supóngase que la entidad *Employee* hace uso de *Field Access* por defecto y que la base de datos utiliza el prefijo del número de teléfono (+33 FR, +49 DE…) como código de localidad para los números

extranjeros. Si para la aplicación únicamente se quisiera obtener el número de teléfono completo cuando este no sea un número local se podría añadir la propiedad (a través del *Property Access*) correspondiente que realice la transformación.

Para definir un *Mixed Access* en este caso, es necesario primero declarar el acceso por defecto:

```
@Entity
@Access(AccessType.FIELD)
public class Employee { … }
```

Y después declarar la excepción de acceso específica para la columna *phone* de la base de datos.

```
@Access(AccessType.PROPERTY) @Column(name = "PHONE")
Protected String getPhoneNumberForDb() { … }
```

Para terminar, se debe declarar el atributo de la clase que pasa de ser un campo de la entidad a una propiedad por medio de la anotación `@Transient`.

```
@Transient private String phoneNum;
```

El código completo quedaría de la siguiente manera.

Ejemplo 3. Uso de *Mixed access*

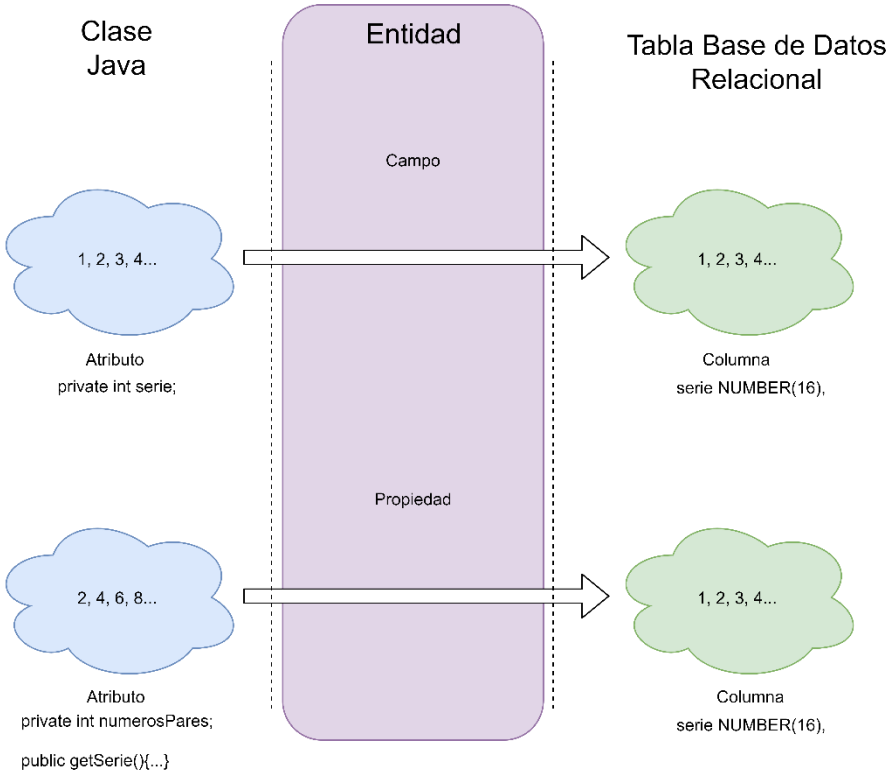
```
@Entity
@Access(AccessType.FIELD)
public class Employee {

 public static final String LOCAL_AREA_CODE = "+34";

 @Id private long id;
 @Transient private String phoneNum;
 …
 public long getId() { return id; }
 public void setId(long id) { this.id = id; }
 public String getPhoneNumber() { return phoneNum; }
 public void setPhoneNumber(String num) { this.phoneNum = num; }

 @Access(AccessType.PROPERTY) @Column(name="PHONE")
 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;
  }
  …
}
```

Como se ha podido ver, este tipo de acceso permite trabajar en Java con números de teléfono completos, respetando el diseño de la base de datos en la cual los teléfonos locales se almacenan sin prefijo y los números extranjeros al completo.

3.2.2. Síntesis

Como se ha visto en el apartado anterior existen sutiles diferencias en la manera de persistir un campo o una propiedad de una entidad. Aun así, se intuye una fuerte equivalencia entre los **atributos** de una Clase y los **campos** o **propiedades** del concepto de Entidad desde el punto de vista de la persistencia como se ilustra a continuación.



En este ejemplo, cuando el conjunto abstracto "serie" (1,2,3,4…) es el mismo en ambos dominios, se considerará el término campo de la entidad como relación entre el atributo y la columna.

Por otro lado, cuando siendo ambos un mismo elemento abstracto, la información se presenta distinta según el dominio en el que se está (*numerosPares* y *serie*), la relación entre el atributo y la columna se conoce como propiedad de la entidad, y su mapeado deberá contener la lógica de la transformación.

Dado que la abstracción es un denominador común y las declaraciones de entidad suceden en el dominio de Java, de ahora en adelante, se hablará de atributos para referirse indiferentemente a campo o propiedad de la entidad.

3.3. Mapeado de clases a tablas

En esta sección se presentará el mapeado de las clases que definen las entidades, que son la relación a las tablas de las bases de datos. También se explicará el mapeo entre los datos de la entidad a las bases de datos, así como las anotaciones existentes para poder realizar dichos mapas.

3.3.1. Primeros conceptos

Se dirá que una entidad está mapeada a una tabla de una base de datos cuando en la clase Java que define la entidad, aparezcan las anotaciones `@Entity` e `@Id`. No obstante, la tabla tiene que existir en la base de datos, así como su identificador. Es en estos casos en los que el nombre por defecto (*default*) de la clase, se intenta identificar con el nombre de la tabla en la base de datos. Si el nombre de la tabla no coincide con el de la entidad, se deberá especificar el nombre real de la tabla en la base de datos. Con ese fin se utiliza la anotación `@Table(name = "NAME")`. De esta manera, es posible crear una clase en Java que tenga un nombre diferente a la tabla de la base de datos a la cual se quiere mapear.

Ejemplo 4. Sobreescribir un nombre de tabla por defecto

```
@Entity
@Table(name="EMP")
public class Employee { ... }
```

En este ejemplo, la entidad se llama *Employee*, pero la tabla de la base de datos a la que hace referencia, se llama *EMP*.

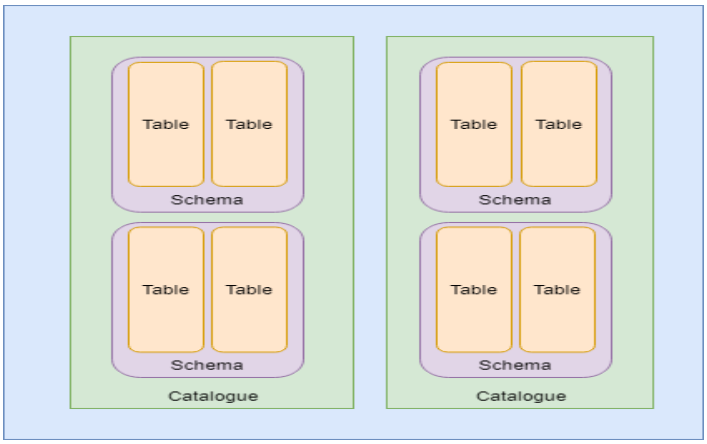
Vale la pena destacar que la mayoría de bases de datos no distinguen entre mayúsculas y minúsculas, de manera que, en general, no importa la manera en la que se escriba el nombre de las mismas.

La anotación `@Table` no solo permite nombrar la tabla donde los datos de la entidad se van a persistir, sino que también permite nombrar el esquema o el catálogo.

Se define esquema (*schema* en inglés) como un conjunto de tablas de una base de datos.

Se llamará catálogo (*catalogue* en inglés) a un conjunto de esquemas.

Una base de datos es el conjunto de esquemas.



Ejemplo 5. @Basic con *Schema* y con *Catalogue*

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }

@Entity
@Table(name="EMP", catalog="HR")
public class Employee { ... }
```

3.3.2. Tipos de datos

En esta subsección se tratará el mapeo de algunos tipos de datos.

Tipos simples

La lista de tipos persistibles es bastante larga:

- Tipos primitivos de Java
- Clases envolventes de los tipos primitivos
- *Arrays* de *bytes* y *char*
- Números grandes
- *Strings*
- Tipos temporales de Java
- Tipos temporales de JDBC
- Tipos enumerados
- Objetos serializables

A veces, el dato que aparece en la columna de la base datos que va a ser mapeado no es exactamente el mismo tipo de dato que hay en la entidad correspondiente en Java. En general, de manera automática,

el tipo retornado por JDBC se convertirá en el mismo tipo que el atributo de la clase de Java. En caso contrario, se obtendrá una excepción.

Cuando se quiere persistir un campo o una propiedad, el proveedor revisa su tipo y se asegura que es uno de los tipos "persistibles" mencionados anteriormente. Si está en la lista, entonces se persiste usando el tipo de JDBC y pasa a través del driver de JDBC. En ese momento, si el campo o la propiedad no es "serializable", el resultado no se especifica.

3.3.3. Anotaciones

En esta subsección se van a presentar algunas de las anotaciones más básicas.

Las anotaciones **son una forma de metadatos** que proporcionan datos sobre un programa. Las anotaciones **no tienen ningún efecto directo** en el funcionamiento del código que anotan.

Como primer ejemplo de anotación está `@Basic`. Esta anotación se puede utilizar para marcar explícitamente que el campo o la propiedad es persistente. No obstante, es utilizada mayoritariamente para temas de documentación.

Mapeo de atributos a columnas

Para mapear los atributos de una clase de Java a una base de datos, se utilizará la anotación `@Column` porque cada uno de ellos se corresponda con una columna de una tabla de una base de datos.

La anotación `@Column` también tiene la función de sobrescribir el nombre por defecto que se le da a un atributo que será mapeado a la base de datos por el nombre que se desee.

Ejemplo 6. Mapeando atributos a columnas

```
@Entity
public class Employee {
  @Id
  @Column(name="EMP_ID")
  private long id;
  private String name;
  @Column(name="SAL")
  private long salary;
  @Column(name="COMM")
  private String comments;
  // ...
}
```

En este ejemplo se puede observar que, en la base de datos, la tabla que está relacionada con la entidad *Employee*, tiene tres columnas que difieren en nombre con los campos de la entidad. Tales columnas son EMP_ID, SAL, COMM. De esta forma, los atributos pueden ser nombrados como se desee o como se necesite, y ser mapeados con las columnas correspondientes en la base de datos gracias a la anotación `@Column`.

Lazy Fetching

Habrá veces en que se sabrá con antelación que ciertos atributos de una entidad no serán requeridos con frecuencia. Obtener cada vez todos los datos de dicha entidad y saturar el mapeo con campos que no suelen ser requeridos, no es una práctica óptima. De ahí nace la idea del *lazy fetching*.

La anotación `@Basic` vista anteriormente, admite como elemento el enumerado *FetchType*. Este enumerado tiene las opciones LAZY y EAGER.

La opción LAZY permite que el proveedor aplace la carga del estado de los datos de cierto atributo hasta que se haga referencia a él.

Para los atributos utilizados frecuentemente se utiliza un retorno del tipo EAGER, ya que siempre carga el estado de los datos de un atributo. Este último es el que se aplica por defecto.

Ejemplo 7. Lazy Fetching

```
@Entity
public class Employee {
  // ...
  @Basic(fetch=FetchType.LAZY)
  @Column(name="COMM")
  private String comments;
  // ...
}
```

Se puede observar que en este ejemplo se está mapeando el atributo `comments` de la entidad a la columna COMM de la base de datos en modo LAZY.

En general, no es buena idea etiquetar con LAZY a los atributos de tipo simple a no ser que no se accede en absoluto a ellos.

Objetos grandes

Se dirá que un objeto es un objeto grande (que se abrevia por *LOB*) si su rango es, como mínimo, de un gigabyte.

Para indicar al proveedor que se debe usar un *LOB* cuando se pasan datos desde o hacia el driver de JDBC, se debe usar la anotación `@Lob` sobre el atributo en cuestión.

Ejemplo 8. Anotación *LOB*

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

Tipos Enumerados

Los tipos enumerados son valores constantes que pueden ser tratados de formas diferentes dependiendo de las necesidades de la aplicación. En Java, el tipo enumerado tiene una asignación ordinal (numérica) implícita que se determina por el orden en el que fueron declarados los elementos de la enumeración. Esta asignación puede ser utilizada para representar y almacenar los valores del tipo enumerado en la base de datos, de forma que el proveedor de persistencia asumirá que la columna en la que se almacenan estos valores son de tipo numérico.

Considerando el siguiente tipo enumerado:

```
public enum EmployeeType {
 FULL_TIME_EMPLOYEE,
 PART_TIME_EMPLOYEE,
 CONTRACT_EMPLOYEE
}
```

Serían asignados el 0 para `FULL_TIME_EMPLOYEE`, el 1 para `PART_TIME_EMPLOYEE` y el 2 para `CONTRACT_EMPLOYEE`. A continuación, se verá un ejemplo de mapeo de un tipo enumerado ordinal:

Ejemplo 9. Mapeado de un tipo enumerado usando ordinales

```
@Entity
 public class Employee {
 @Id private long id;
 private EmployeeType type;
 // ...
}
```

En este ejemplo, el campo *type* será mapeado a una columna de tipo *Integer* sin problemas. Sin embargo, si alguno de los elementos de la enumeración cambia, sí que ocurrirán problemas debido a que los datos numéricos persistidos en la base de datos ya no se corresponderán con los nuevos valores asignados. Por ejemplo, si después de `PART_TIME_EMPLOYEE` se añadiese otro elemento, todos los

empleados con contrato pasarían a ser del nuevo tipo de empleados en la base de datos, que claramente no es el resultado deseado ya que implicaría ir a la base de datos a cambiar la entidad *Employee.*

Otra solución sería no guardar el ordinal, sino el mismo valor como *String* utilizando el elemento `EnumType` con la opción *Stirng* de la anotación `@Enumerated`:

Ejemplo 10. Mapeado de un tipo enumerado usando `String`

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

Esta forma de actuar causaría problemas a la hora de cambiar el nombre de los elementos de la enumeración, provocando el cambio de todo el código que haga uso del tipo enumerado.

En definitiva, la manera más eficiente de almacenar este tipo de datos es mediante los ordinales, pero teniendo en cuenta que, si se desea añadir nuevos elementos, se debe hacer al final.

Tipos Temporales

Los tipos temporales son el conjunto de los tipos de datos basados en fechas, que pueden usarse en el mapeo de la entidad. Los tres tipos de datos que definen el conjunto de tipos temporales de `java.sql types` son: `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp` y lo dos de `java.util` son `java.util.Date`, `java.util.Calendar`.

La anotación para estos tipos es `@Temporal` y se debe especificar a JDBC el tipo de valor del enumerado TemporalType que tiene por argumento la anotación. Los valores del enumerado son: DATE, TIME, TIMESTAMP.

Ejemplo 11. Tipos Temporales

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

```
  // ...
}
```

Estado Transitorio (*Transient)*

Los campos transitorios de una entidad son aquellos que no participan en el proceso de persistencia y, por tanto, sus valores nunca se guardan en la base de datos.

La sintaxis de Java admite dos alternativas para dotar de la característica *Transient* a un campo. Sin embargo, ambas opciones no son equivalentes.

- La primera manera es con la anotación `@Transient`. Esta indica que el campo no debe ser persistido en la base de datos.
- La segunda opción es usando la palabra reservada `transient`. Esta opción se usa para denotar que el campo **no** es serializable.

Ejemplo 12. Tipos *transient*

```
@Entity
public class Employee {
  @Id private long id;
  private String name;
  private long salary;
  @Transient private String translatedName;
// ...
}
```

En este ejemplo se ha utilizado la anotación para especificar que el atributo `translatedName` es transitorio.

3.3.4. Primary Key

Cada clase entidad que es mapeada a una base de datos relacional debe tener un atributo mapeado a una PRIMARY KEY en la tabla. En esta sección se van a ver los identificadores simples (compuestos por un solo atributo) y las PRIMARY KEYS con más profundidad que en las secciones anteriores. Además, se explicará la manera en la que se puede hacer que el proveedor de persistencia genere valores identificadores únicos.

Cuando se trata del atributo que se asocia a la columna referente a la PRIMARY KEY, como esta no puede ser *null* ni modificable, no es nada recomendable declarar ese atributo como *nullable* o *updatable*.

Tipos de PRIMARY KEY

Los tipos de datos en Java a los que está restringida la PRIMARY KEY son primitivos, envolventes, `String`, para números grandes `java.math.BigInteger` y para fechas `java.util.Date` y `java.sql.Date`. Utilizar tipos de datos de coma flotante no es recomendable.

Generación de identificadores

A menudo las aplicaciones dejan que los valores identificadores se generen automáticamente. Esto se denomina *ID generation* y es especificado por la anotación `@GeneratedValue`. Cuando este es habilitado, el proveedor de persistencia genera un valor identificador para cada instancia de esa entidad. Sin embargo, es posible que el objeto de Java no reconozca el identificador hasta que no se haya guardado en la base de datos (o bien mediante el método *flush,* o bien una vez se haya terminado la transacción).

Los generadores de tablas y secuencias pueden ser definidos específicamente y luego reutilizados por múltiples clases entidad. Estos generadores son globalmente accesibles a todas las entidades de la unidad de persistencia.

Existen cuatro estrategias para la generación de identificadores (especificando el valor del elemento `stategy` de `@GeneratedValue`). Estos son:

- ***AUTO***: Si no interesa especificar el tipo de generación, se puede utilizar la forma automática. De esta forma, será el proveedor de persistencia el que elija cuál de las otras estrategias usar. Esta estrategia se utiliza únicamente para desarrollar o hacer prototipos porque para que funcione es necesario tener permisos para crear tablas o secuencias en la base de datos (que es lo que hacen las otras estrategias existentes).

  Ejemplo 13. Uso del *Auto ID Generation*

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

  Tal como se muestra en el ejemplo, es recomendable utilizar el tipo de variable *long* para el identificador, de manera que se pueda aprovechar toda la extensión del dominio del identificador generado.

- ***TABLE:*** La utilización de una tabla de la base de datos es la forma más flexible para generar identificadores. Esta estrategia permite ser utilizada en diferentes bases de datos, y también almacenar diferentes identificadores de secuencias para las PRIMARY KEYS de distintas entidades en la misma tabla.

Una tabla de *ID generation* debería tener dos columnas. La primera columna de tipo *Stirng* se utiliza para identificar las diferentes secuencias del generador. Es como una clave primaria para todos los generadores de la tabla.

La segunda columna es de tipo numérico y almacena el último número generado por la secuencia. Cada generador se corresponde con cada fila de la tabla.
La forma más simple de indicar el tipo de estrategia sería la siguiente:

```
@Id @GeneratedValue(strategy=GenerationType.TABLE)
private long id;
```

Sin embargo, no es la más adecuada, ya que no se le indica cuál es la tabla de *ID generation,* así que el proveedor de persistencia tomará cualquiera. Por lo tanto, es mejor indicar la estrategia utilizando una anotación referida a la misma, de la siguiente forma:

```
@TableGenerator(name="Emp_Gen")
@Id @GeneratedValue(generator="Emp_Gen")
private long id;
```

En la primera anotación `@TableGeneration`, se indica, mediante el elemento `name`, el nombre del generador de la tabla, lo que anteriormente se explicó que se indicaba en la primera columna de la misma. Esto permite utilizar este nombre en el elemento `generator` de la anotación `@GeneratedValue`.

Otra forma de expresarlo con más detalle sería indicando el nombre del generador, el de la tabla, el de la columna PRIMARY KEY y el de la columna de valores:

```
@TableGenerator(name="Emp_Gen",
table="ID_GEN",
pkColumnName="GEN_NAME",
valueColumnName="GEN_VAL")
```

La tabla en la base de datos sería la siguiente, donde los nombres que se acaban de indicar en el ejemplo coinciden con los de la misma:

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0

Para evitar actualizar la fila para cada identificador que se genere, el proveedor de persistencia utiliza un bloque de identificadores preasignados hasta que se agota. Una vez agotado este bloque, la siguiente solicitud de identificador desencadena otro bloque. De forma

predeterminada, el tamaño del bloque es de 50. Este valor puede ser sustituido para ser mayor o menor mediante el uso del elemento `allocationSize` al definir el generador.

El proveedor de persistencia puede estar identificando a la vez que persistiendo entidades o hacerlo en transacciones separadas. No se especifica, pero ha de comprobarse en las características del proveedor para ver cómo se puede evitar el conflicto de creación de entidades a la vez que se actualizan lo registros.

Un ejemplo con dos generadores sería el siguiente:

Ejemplo 14. Uso de *ID Generation*

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0
Addr_Gen	10000

```
@TableGenerator(name="Address_Gen",
table="ID_GEN",
pkColumnName="GEN_NAME",
valueColumnName="GEN_VAL",
pkColumnValue="Addr_Gen",
initialValue=10000,
allocationSize=100)
@Id @GeneratedValue(generator="Address_Gen")
private long id;
```

La creación de la tabla en SQL se haría se la siguiente forma:

```
CREATE TABLE id_gen (
gen_name VARCHAR(80),
gen_val INTEGER,
CONSTRAINT pk_id_gen
PRIMARY KEY (gen_name)
);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Emp_Gen', 0);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Addr_Gen', 10000);
```

- ***SEQUENCE:*** Una secuencia de la base de datos puede ser utilizada para generar identificadores. Si solo se utiliza una o no es importante saber cuál de ellas se utiliza cada vez, es suficiente con indicar la estrategia en la anotación `@GeneratedValue`:

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
private long id;
```

Y si no, habría que indicar el nombre del generador y de la clase de la siguiente forma:

```
@SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
@Id @GeneratedValue(generator="Emp_Gen")
private long getId;
```

Para generar la secuencia en SQL se seguirían las siguientes instrucciones:

```
CREATE SEQUENCE Emp_Seq
MINVALUE 1
START WITH 1
INCREMENT BY 50
```

- ***IDENTITY:*** Algunas bases de datos tienen una PRIMARY KEY que, cada vez que una fila es insertada a una tabla, genera un identificador único asignado a esta. Esta puede ser empleada para generar identificadores para objetos de Java, pero no todas las bases de datos tienen esta funcionalidad. Al generarse desde la base de datos, no está disponible hasta que la instancia de la entidad no sea insertada en la base de datos y hasta que no se haya utilizado el método *commit* para dar por completada la transacción (el *commit time)*, lo que no es muy eficiente. La estrategia se indicaría se la siguiente forma:

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private long id;
```

Así se indicaría al proveedor de persistencia que debe volver a procesar la fila insertada en la tabla de la base de datos después de haberla insertado para recuperar el identificador que se ha generado y ponerlo en memoria de la entidad que acaba de ser persistida.
Por otro lado, no hay anotación para esta estrategia en particular porque esta no puede ser utilizada para múltiples tipos de entidades.

Cabe destacar que algunos proveedores insertan de manera *eagerly* (cuando se invoca el método *persist*) entidades que están configuradas para utilizar la generación de IDENTITY ID, en lugar de esperar hasta el *commit time*. Esto permitirá que el ID esté disponible inmediatamente. Por eso es importante conocer la forma de actuar del proveedor que se está utilizando.

3.4. Relaciones

Hasta ahora se ha visto cómo definir las relaciones entre una clase y una tabla para definir la entidad. Una base de datos no sólo almacena la información física de los *employees* y *departments* en sus tablas, sino que también contiene información acerca de las relaciones entre distintas tablas. Es por eso que en este apartado se explica cómo se definen las relaciones entre distintas entidades, por medio del mapeado de las asociaciones entre clases.

Para ello es oportuno exponer el concepto de **roles** en una relación y los elementos que derivan de ellos: **direccionalidad**, **cardinalidad** y **ordinalidad**.

3.4.1. Conceptos

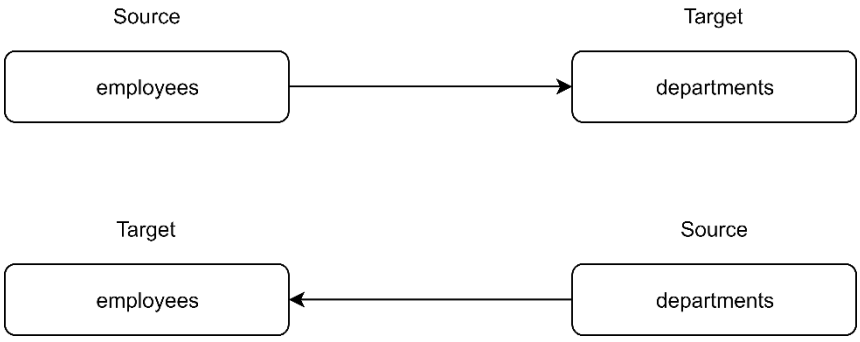
Roles y Direccionalidad

Las relaciones entre entidades tienen tres puntos de vista: una desde un lado de la relación, otra desde el otro lado, y una perspectiva global vista desde fuera. Los "lados" de la relación son los llamados roles. En toda relación hay dos entidades asociadas entre sí y cada una desempeña un rol. Una entidad puede estar implicada en varias relaciones y, es posible, que desempeñe diferentes roles dentro de un mismo modelo.

Una relación entre entidades se establece al incluir un atributo a la entidad que aluda a la otra entidad. De esta forma, si sólo una entidad de la relación contiene el atributo que alude a la otra, la relación es unidireccional, en el que esta entidad tiene el rol de *source* (origen) y la otra el rol de *target* (destino).

Si por el contrario ambas entidades contienen atributos que aluden a la otra, la relación es bidireccional.

Es el punto de vista el que define los roles, por lo que cualquier relación bidireccional es la suma de dos relaciones unidireccionales.



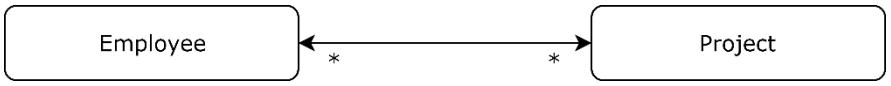
Por ejemplo, un empleado que conoce el proyecto en el que trabaja, y un proyecto que conoce qué trabajadores trabajan en él. Cada entidad es *source* y *target* de cada relación unidireccional que forman la bidireccional:

Un ejemplo de relación unidireccional sería la de un empleado que conoce su dirección, pero el empleado no es conocido por la dirección:

Cardinalidad

La cardinalidad de una relación es la forma en la que se relacionan las entidades. Cada rol de la relación tiene una cardinalidad, y esta expresa cuántas entidades se relacionan con otras entidades, si es una o más de una. Por ejemplo, si se afirmara que un empleado trabaja en un departamento se tendría una cardinalidad de uno en los dos roles de la relación, ya que la entidad *Department* se relaciona solo con una entidad, la entidad *Employee,* y viceversa. Sin embargo, si pudieran trabajar más de un empleado en un mismo departamento, la cardinalidad de la *source*, es decir, del rol de *Employee,* pasaría a ser múltiple, mientras que la de *Department* seguiría siendo uno. En UML, la cardinalidad múltiple se indica con un asterisco (*). Por lo tanto, se tendría una relación de muchos a uno, lo que se denomina *many-to-one.*

Un ejemplo de *many-to-many* sería el mencionado anteriormente de *Employee* y *Project*, ya que cada empleado puede pertenecer a muchos proyectos, así como varios empleados pueden formar parte de un mismo proyecto. La cardinalidad se representaría de la siguiente forma:



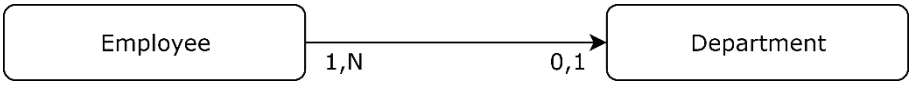
Ordinalidad

La ordinalidad es un valor booleano que sirve para indicar si una relación es opcional o no. Es decir, si es obligatorio especificar o declarar la entidad *target* cuando se crea una de la entidad *source*.

En términos de cardinalidad, la ordinalidad se indicaría dándole un rango a la cardinalidad, es decir, determinando si el rango de la cardinalidad empieza en 0 o en 1. Volviendo al ejemplo de *Employee* y *Department*, la ordinalidad determinaría dos aspectos: en primer lugar, si es posible que un

departamento no tenga ningún empleado o si en cambio, para que exista un departamento es necesario que exista como mínimo un empleado en él. En segundo lugar, si es posible que un empelado no tenga asignado ningún departamento, o si, por el contrario, es imposible tener un empleado que no pertenezca a ningún departamento en la empresa.

El ejemplo según el cuál sería necesario como mínimo un empleado para tener un departamento y pudiesen haber empleados sin departamento se podría representar de la siguiente forma:



3.4.2. Mapeado de relaciones entre entidades

En el mapeo de relaciones entre entidades existen diferentes anotaciones posibles en función de las combinaciones de cardinalidad entre el *source* y el *target*, en ese orden:

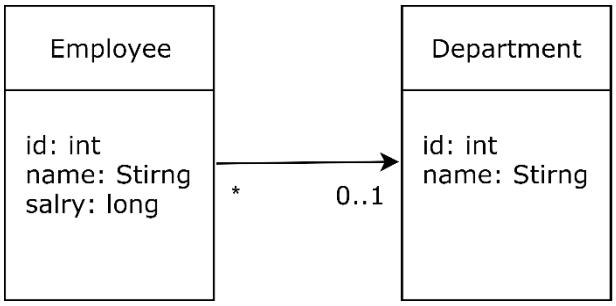
- *@ManyToOne*
- *@OneToOne*
- *@OneToMany*
- *@ManyToMany*

Mapeados ManyToOne y OneToOne

Estos dos tipos de mapeados tienen en común que la entidad *source* refiere a, como máximo, una sola entidad *target.*

- **ManyToOne**

Se puede destacar que en el diagrama UML, la clase *source* tiene un atributo implícito del tipo de clase *target* si se puede navegar hacia ella. Por ejemplo, haciendo referencia al ejemplo de relación entre *Employee* y *Department* mostrado en la siguiente figura, *Employee* tiene un atributo llamado *department* que contendrá una referencia a una única instancia de *Department*. Este atributo no se muestra en la clase *Employee*, pero está implícito por la presencia de la flecha de relación.



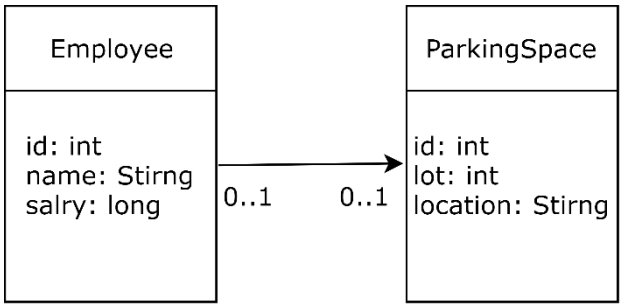
Un ejemplo de la utilización de la anotación:

Ejemplo 14. Relación *ManyToOne* de *Employee* a *Department*

```
@Entity
 public class Employee {
 // …
 @ManyToOne
 Private Department department;
 //…
}
```

- **OneToOne**

Un buen ejemplo de la relación *OneToOne* sería la que hay entre un empleado y su plaza de parking, asumiendo que cada empleado tiene una asignada. Esta se puede ver en la siguiente figura:



El mapeo se hace igual que el de *ManyToOne* pero con la anotación `@OneToOne` y teniendo en cuenta que solo una instancia de la entidad *source* puede referirse a la misma entidad *target.* En la base de datos, esto equivale a que la FOREIGN KEY de la tabla que se refiere a la entidad *source*, tendrá la restricción UNIQUE.

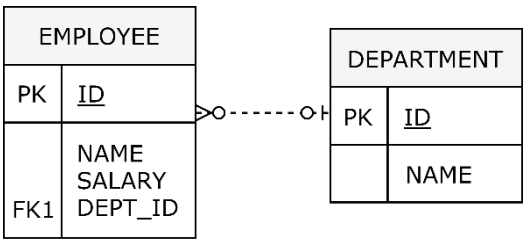
Un ejemplo de la utilización de la utilización de la anotación:

Ejemplo 15. Relación *OneToOne* de *Employee* a *ParkingSpace*

```
@Entity
public class Employee {
 @Id private long id;
 private String name;
 @OneToOne
 private ParkingSpace parkingSpace;
 // ...
}
```

Anotaciones relevantes

Se utiliza la anotación `@JoinColumn` para configurar las columnas FOREIGN KEY que hacen referencia a una clave (normalmente la PRIMARY KEY) de otra tabla. En la relación entre *Employee* y *Department* esta columna, que asocia ambas entidades, sería DEPT_ID:



En casi todas las relaciones, independientemente de si se trata de la parte del *source* o del *target*, una de las dos partes tendrá la columna de unión en su tabla. Ese lado de la relación se llama el lado propietario o propietario de la relación. El lado que no tiene la columna de unión se llama lado no propietario.

La propiedad es importante para el mapeo, ya que las anotaciones que definen los mapeos a las columnas de la base de datos (por ejemplo, @*JoinColumn*) son siempre definidas en el lado propietario de la relación. Si no están ahí, los valores por defecto estarán en el lado que favorezca la dirección de mapeo más frecuente.

Los mapeos *ManyToOne* siempre están en el lado propietario de la relación, así que, si hay una `@JoinColumn` en esta relación, estará en este lado. Para especificar el nombre de la columna de unión, se utiliza el elemento de configuración *name*.

Por ejemplo, la anotación `@JoinColumn(name="DEPT_ID")` significa que la columna DEPT_ID en la tabla de entidad *source* es la *foreign key* de la tabla de entidad *target*, sea cual sea la entidad *target* de la relación.

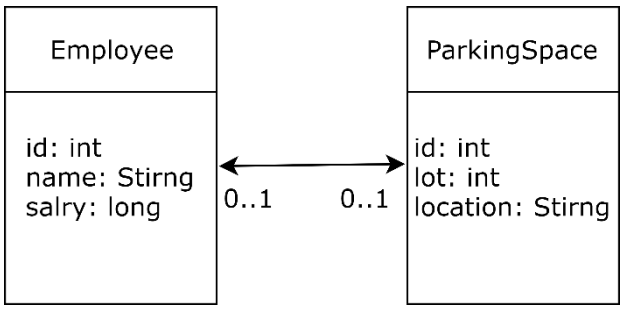
Si no hay ninguna anotación `@JoinColumn` que acompañe el mapeo de *ManyToOne*, se asumirá un nombre de columna por defecto: el nombre del atributo de relación en la entidad *source*, que es *department* en nuestro ejemplo, más un guión bajo (_), más el nombre de la columna *primary key* de la entidad *target*. Por lo tanto, si la entidad *Department* fuera mapeada a una tabla que tuviera una columna *primary key* denominada ID, se asumiría que el nombre de la columna de unión en la tabla EMPLOYEE es, por defecto, DEPARTMENT_ID. Si su nombre es otro, este debe indicarse haciendo uso del elemento *name* de la anotación `@JoinColumn`.

Volviendo a la anterior figura, la columna *foreign key* se llama DEPT_ID en lugar del nombre que se asignaría por defecto, DEPARTMENT_ID. El ejemplo 16 muestra la anotación `@JoinColumn` que se utiliza para sobreescribir el nombre de la columna de unión por DEPT_ID. Se ha escrito en primer lugar la anotación `@ManyToOne` y en segundo `@JoinColumn` porque es la forma más lógica de hacerlo, sin embargo, estaría permitido invertir el orden.

Ejemplo 16. Relación *ManyToOne* sobreescribiendo la columna de unión

```
@Entity
public class Employee {
@Id private long id;
@ManyToOne
@JoinColumn(name="DEPT_ID")
private Department department;
// ...
}
```

Por otro lado, es necesario incluir en el código, en la clase que hace referencia al lado no propietario de la relación, el elemento *mappedBy* de la anotación `@OneToOne` o `@ManyToOne`. De lo contrario, se asumirá que se está en el lado propietario de la misma. Este elemento será igual al nombre que se le ha dado, en la clase que hace referencia al lado propietario de la relación, al atributo referido al lado propietario de la misma. Para visualizarlo mejor, se retoma el ejemplo del empleado y la plaza de parking, pero cambiando la relación *OneToOne* unidireccional, a una *OneToOne* bidireccional. Este cambio de concepción del modelo se haría si la plaza tuviera una referencia al empleado que la posee, y no únicamente fuera el empleado el que tuviera una referencia a la plaza que le pertenece.



Como se ha dicho anteriormente, la tabla entidad que contenga la columna de unión será la considerada propietaria de la relación. Sin embargo, en una relación bidireccional cualquiera de los lados puede ser el propietario. Es una decisión de modelado, no de programación en Java, la de elegir qué lado será el propietario. En el ejemplo que se está tratando, se asume que este será el empleado. Como se ha visto, esto derivará en la existencia de una referencia a *Employee* dentro de la clase *ParkingSpace*, se utilizará para ello la anotación `@OneToOne`. Pero eso no es todo, se deberá incluir además el elemento *mappedBy* para indicar que la parte propietaria de la relación es *Employee* y no *ParkingSpace*.

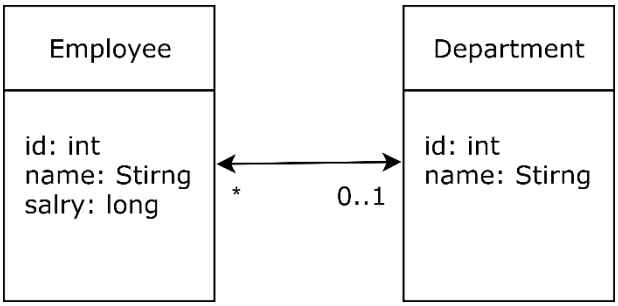
Ejemplo 17. Parte no propietaria de una relación *OneToOne bidireccional*

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

Se observa que la columna de unión no se encuentra en la entidad a la que hace referencia el ejemplo y que con el elemento de *mappedBy* se hace referencia al atributo *parkingSpace* que existe en la clase *Employee.*

OneToMany y ManyToMany

Son anotaciones que se utilizan cuando la entidad *source* hace referencia a una o más instancias de la entidad *target*. Aunque el mapeo *OneToMany* sea el más utilizado, el *ManyToMany* es muy útil cuando se comparten en ambas direcciones. A continuación, se verán algunos ejemplos:



En la figura se puede observar una relación que, según si se ve desde *Employee* es *ManyToOne,* o *OneToMany* si se ve desde *Department.* Ambas relaciones son equivalentes porque al relacionarse bidireccionalmente, la existencia de una implica la existencia de la otra.

En este tipo de relaciones, se decide casi siempre que el propietario de la relación sea la parte *one* del *OneToMany.* De esta forma es posible guardar las FOREIGN KEYS que hacen referencia a la entidad *source,* en las entidades *target.* A continuación, se completará el ejemplo 14 (que mostraba *Employee*) con el código que tendría la clase *Department:*

Ejemplo 18. Relación *OneToMany*

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

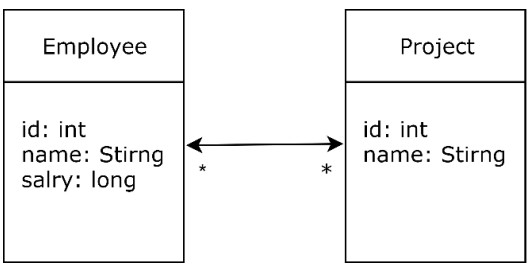
Como esta sería la parte no propietaria de la relación, se utiliza *mappedBy*, al igual que en el Ejemplo 17. Además, como a cada departamento pertenecen varios empleados, se utiliza el tipo genérico parametrizad *Collection* para almacenar las entidades de los Empleados, esta garantiza que solo los objetos de tipo *Employee* formarán parte de la misma. Sin utilizar genéricos, podría hacerse también de la siguiente forma:

Ejemplo 19. Uso de *targetEntity*

```
@Entity
public class Department {
 @Id private long id;
 private String name;
 @OneToMany(targetEntity=Employee.class, mappedBy="department")
 private Collection employees;
 // ...
}
```

Al no especificar el tipo de dato que admite `Collection`, es necesario indicar de otra forma el tipo de entidad a persistir, esto es, con el elemento `targetEntity` de `@OneToMany`.

A continuación, se verá un ejemplo de *ManyToMany.* Esta relación se da cuando una o más entidades están asociadas con una *Collection* de otras entidades y las entidades tienen asociaciones superpuestas con las mismas entidades *target*. La figura que se muestra a continuación ilustra este tipo de relación entre *Employee* y *Project* ya que cada empleado puede trabajar en múltiples proyectos y cada proyecto puede tener implicados a múltiples empleados:



Ejemplo 20. Relación *ManyToMany* entre *Employee* y *Project*

```
@Entity
public class Employee {
 @Id private long id;
 private String name;
 @ManyToMany
 private Collection<Project> projects;
 // ...
}

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

Se observa que, evidentemente, las dos entidades son mapeadas como *ManyToMany* al ser bidireccional. También es interesante notar que no hay columnas de unión en ningún lado de la relación. A causa de esto se hace imposible averiguar cuál de los dos lados es el propietario de la relación. Porque toda relación bidireccional debe tener tanto un lado propietario y un lado no propietario o inverso, se debe elegir una de las dos entidades como propietaria. En este ejemplo, se elige *Employee* para ser el propietario de la relación, pero podría haber sido elegido, de igual manera, *Project.* Como en cualquier otra relación bidireccional, el lado inverso debe utilizar el elemento *mappedBy* para identificar el atributo de propiedad.

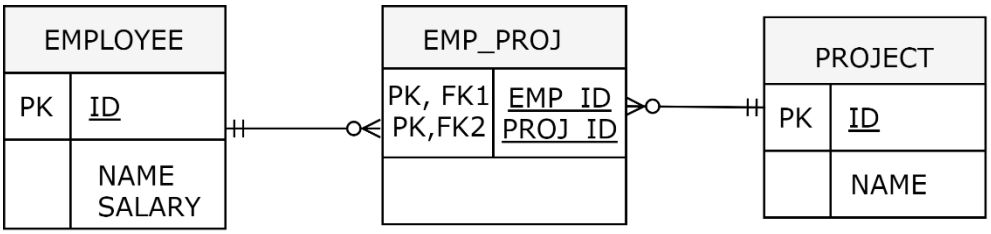
Es importante tener en cuenta que, independientemente de qué parte se designe como propietaria, la otra parte deberá incluir el elemento *mappedBy*; de lo contrario, el proveedor pensará que ambos lados son propietarios y que los mapeos son relaciones unidireccionales separadas. Esto mismo ocurrirá si no se incluye este elemento en una relación *OneToMany*. Para que exista la relación que se pretende, es necesaria una tabla de asociación. Si es para una relación de *OneToMany*, solo una de las dos entidades usa la tabla para cargar sus entidades relacionadas. En cambio, para una *ManyToMany*, las dos la utilizarían. El uso de tablas de asociación se explica a continuación.

Anotaciones relevantes

Ya que la multiplicidad de ambos lados de una relación *ManyToMany* es plural, ninguna de las dos tablas entidades puede almacenar un conjunto ilimitado de valores de FOREIGN KEY en un solo registro de una entidad. Es necesario usar una tercera tabla para asociar los dos tipos de entidades. Esta tabla de asociación es llamada tabla de unión, y cada relación *ManyToMany* requiere de una. Podrían ser utilizadas también para los otros tipos de relación, pero no son necesarias.

Una tabla de unión consiste simplemente en dos FOREIGN KEYS o columnas de unión que se refieren a cada uno de los dos tipos de entidades en la relación. Una colección de entidades es entonces mapeada como múltiples filas de la tabla, cada una de las cuales asocia una entidad con otra. El conjunto de filas que contienen un identificador de entidad dado en la columna de FOREIGN KEY de la entidad *source* de la relación, representan la colección de entidades relacionadas con esa entidad.

La siguiente figura muestra las tablas EMPLOYEE y PROJECT para las entidades *Employee* y *Project*, y la tabla de asociación EMP_PROJ que las relaciona. Esta última contiene sólo las columnas FOREIGN KEY que componen su PRIMARY KEY compuesta. La columna EMP_ID se refiere a la PRIMARY KEY de EMPLOYEE, mientras que la columna PROJ_ID se refiere a la PRIMARY KEY de PROJECT.



Para mapear las tablas descritas en la figura anterior, es necesario añadir metadatos (*metadata*) a la clase *Employee*, ya que era el que se había decidido asumir como propietario de la relación. Esto se ve a continuación:

Ejemplo 21. Uso de una tabla de asociación

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

La anotación `@JoinTable` se utiliza para configurar la tabla de asociación para la relación. Las dos columnas de unión de la tabla de asociación se distinguen por medio de la parte propietaria y la inversa. Dentro de esta anotación, mediante el elemento `joinColumns`, se describe la columna de unión de la parte propietaria; mientras que mediante el elemento `inverseJoinColumns` se especifica la columna de unión al lado inverso.

Si se estuviera generando el esquema de la base de datos de las entidades, así como no sería necesario especificar el nombre de las columnas de unión, tampoco lo sería el de la tabla. Esta tomaría el nombre

por defecto:<Propietaria>_<Inversa>, donde ambos nombres serían los de la entidad propietaria e y inversa respectivamente. Por supuesto, el propietario es básicamente elegido al azar por el desarrollador, por lo que estos valores predeterminados se aplicarán de acuerdo con la forma en que se asigna la relación y la entidad que se designe como parte propietaria.

Lazy Relationships

En secciones anteriores se ha visto que *lazy loading* a nivel de atributos no es muy usada. Sin embargo, a nivel de la relación, puede ser de gran ayuda para mejorar el rendimiento. Puede reducir la cantidad de código de SQL que se ejecuta y acelerar las consultas y la carga de objetos.

El *fetch mode* se puede especificar en cualquiera de los cuatro tipos de mapeo de relaciones. Cuando no se especifica en una relación *ManytoOne* o *OnetoOne*, se garantiza que el objeto relacionado será *loaded eagerly*. Las relaciones *OneToMany* y *ManytoMany* son por defecto *lazily loaded*, pero también pueden ser *eagerly loaded* si el proveedor así lo decide.

En los casos de relaciones bidireccionales, el *fetch mode* puede ser *lazy* en una dirección, y *eager* en la otra. Este tipo de configuración es bastante común porque a menudo se accede a las relaciones de diferentes maneras dependiendo de la dirección desde la cual se produce la navegación.

Un ejemplo de cómo sobreescribir el *fetch mode* por defecto sería si no se quisiese cargar el archivo *ParkingSpace* para un *Employee* cada vez que se carga el *Employee*. El ejemplo 24 muestra el atributo *parkingSpace* configurado para utilizar la *lazy loading*.

Ejemplo 22. Cambiando el *Fetch Mode* en una relación.

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

3.5. Objetos embebidos

En ocasiones, cuando se trabaja con herramientas de mapeo objeto-relacional, puede ser interesante la inclusión de entidades dentro de otras entidades de forma que las dos (o varias entidades) **compartan una única identidad**. Dichos objetos se llamarán *objetos embebidos*.

En esta sección se presentará el concepto de dichos objetos. Así mismo, se mostrarán algunos ejemplos de aplicaciones de los mismos con su correspondiente sintaxis en Java.

En el párrafo anterior, se ha presentado la idea intuitiva de lo que son los objetos embebidos. No obstante, es necesaria una definición formal:

Aquellos objetos de Java que son dependientes de una entidad para su identidad, i.e., que están contenidos[1] *dentro de los atributos de la entidad, serán llamados* ***objetos embebidos*** *(Embedded Objects).*

A partir de la definición anterior se puede observar que los objetos integrados **no son entidades**, sino que son simplemente objetos de Java que se utilizan para almacenar cierta información de dicha entidad.

Ejemplo 23. Un primer objeto embebido

Supóngase que se tiene la entidad *Employee* con atributos *id, name, salary, street, city, state* y *zip_code*. Extrayendo los atributos de esta entidad que están relacionados con la localización, se puede crear un objeto embebido llamado *Address*. Es decir, el objeto *Address* que tiene por atributos *street, city, state y zip_code* es un objeto embebido de la entidad *Employee*.

En cierto modo, parece que exista una relación entre un objeto embebido y una entidad. No obstante, un objeto embebido **no** está relacionado con una entidad, pues el término de relación únicamente puede ser aplicado cuando ambas instancias relacionadas (*source* y *tarjet*) son entidades.

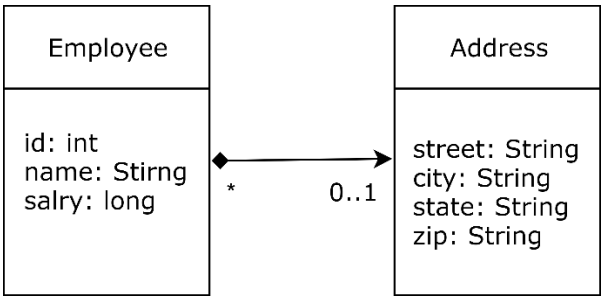
Hasta ahora se ha explicado cual es el vínculo entre un objeto embebido y una entidad en Java, pero, como cabe esperar, también hay un vínculo entre un objeto embebido y una tabla en base de datos. Desde el punto de vista de base de datos, el estado de un objeto embebido y el estado de una entidad se guardan en una misma columna, sin distinción entre el objeto embebido y la entidad de Java. En otros términos, en una base de datos, una entidad y un objeto **son indistinguibles**.

La necesidad de la separación explícita de una entidad con respecto a un objeto embebido se encuentra en el concepto *impedance mismatch* (ver Capítulo 1). Debido a que, cada registro de la base de datos contiene más de un tipo de datos, surge la necesidad de la separación en Java.

La decisión, pues, de usar objetos embebidos o entidades depende de si va a ser necesario crear relaciones entre ellas o no. Si un objeto embebido es tratado como una entidad, entonces se debe plantear si considerarlo como una entidad, o en su defecto, cambiar la estrategia que se ha usado con él. Hay que tener en cuenta que no es eficiente definir objetos embebidos como parte de una jerarquía de herencias. La extensión de objetos embebidos incrementa la complejidad.

A partir del ejemplo 23, se obtiene el esquema ULM con las tablas EMPLOYEES y ADDRESS.

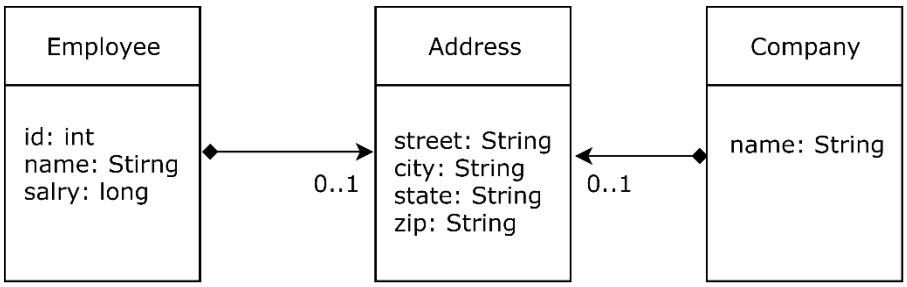
[1] Un objeto esta contenido dentro de una entidad produciéndose una asociación por agregación o composición.



Gracias al ULM del ejemplo anterior, se puede ver que puede haber un empleado sin una dirección, no obstante, no se puede dar que haya una dirección sin estar asociada a algún empleado. De hecho, se puede dar el caso que un mismo objeto embebido sea compartido por varias entidades que compartan la misma representación interna. No obstante, **para cada entidad existe de manera única un objeto embebido**. Esto es, aunque dos entidades diferentes compartan un mismo objeto embebido, para cada una de ellas, el objeto es único; por tanto, el objeto embebido de la primera tabla es independiente al objeto embebido de la segunda tabla, aunque su representación sea la misma.

Ejemplo 24. Esquema UML de dos entidades compartiendo un objeto embebido

Se supone que además de la entidad *Employee,* se tiene otra entidad *Company* tal que ambas entidades comparten el objeto embebido *Address*. En este supuesto, el diagrama ULM es:



A nivel de código Java, un objeto embebido es marcado por la etiqueta *@Embeddable* en la definición de la clase. Con esta anotación se está distinguiendo la clase que define un objeto embebido de una clase regular de Java. Una vez que se ha marcado esta clase como incrustada (*Embedded)*, sus objetos serán persistibles con relación a sus entidades.

Ejemplo 25. Clase embebible *Address*

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

Observación: En el ejemplo anterior, se ha usado el tipo de acceso FIELD, no obstante, también funciona el acceso por PROPERTY.

Una vez se tiene una clase capaz de generar objetos embebidos, se debe crear (al menos) una entidad a la cual dichos objetos estén ligados.

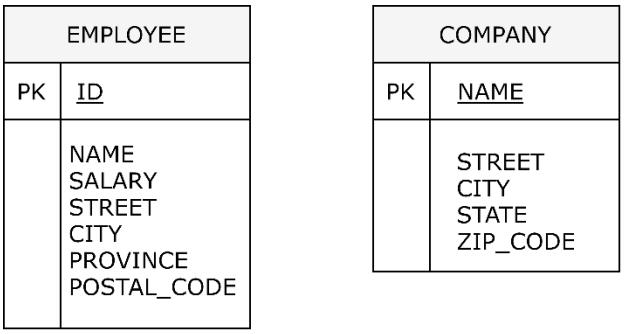
Ejemplo 26. Asociación de un objeto embebido a una entidad

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

En el ejemplo anterior, cuando el proveedor persiste una instancia de *Employee*, inmediatamente tendrá acceso a los atributos del objeto embebido *Address* como si estos fueran de la entidad *Employee* explícitamente.

Podría darse el caso en el que dos entidades tuvieran diferente nombre para un mismo campo. Para solucionar esta discrepancia existe una anotación de Java, `@AttributeOverride`. Esta se usa para especificar el nombre del campo o propiedad del objeto embebido que se está sobrescribiendo en la entidad.

Ejemplo 27. Entidades con diferentes nombres para el mismo campo



En este ejemplo, se puede observar que ambas tablas tienen columnas que hacen referencia al mismo dato: (STREET, STREET*), (*CITY, CITY*), (*PROVINCE, STATE*)* y *(*POSTAL_CODE, ZIP_CODE*)*. A pesar de que, en algunos casos, los nombres de las columnas no son coincidentes, la información que contienen, en esencia, sí lo es.

Ejemplo 28. Asociación de un objeto embebido a dos entidades con diferentes nombres para un mismo campo

```
@Entity
public class Employee {
        @Id private long id;
        private String name;
        private long salary;
        @Embedded
        @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
        })
        private Address address;
        // ...
}
@Entity
public class Company {
        @Id private String name;
        @Embedded
        private Address address;
// ...
}
```

4. JP QL & Query Language

El mapeo objeto-relacional añade una nueva complejidad a la hora de realizar operaciones que impliquen el traslado de datos desde la base de datos a la aplicación. La mayor parte del tiempo, el desarrollador querrá los resultados de una consulta convertidos a entidades para que puedan ser manejados directamente por la lógica de la aplicación. De manera similar, si el modelo de dominio ha sido abstraído del modelo físico vía mapeo objeto-relacional, también tiene sentido abstraer las consultas fuera de SQL ya que, no sólo está ligado al modelo físico, sino que también es difícil de transportar entre proveedores. Afortunadamente, JPA puede manejar un conjunto diverso de requisitos de consulta. JPA tiene dos métodos para expresar consultas y recuperar entidades y otros datos persistentes de la base de datos: *query languages* y el *Criteria API*. El *Java Persistence Query Language* (JP QL) es el lenguaje de consultas estándar de JPA. Es un lenguaje portátil diseñado para combinar la sintaxis y la semántica simple de consultas de SQL con la expresividad de un lenguaje orientado a objetos.

Para describir lo que es JP QL, es importante dejar claro lo que no es. JP QL no es SQL. A pesar de las similitudes entre los dos lenguajes en términos de sentencias y estructura, hay diferencias muy importantes. Las similitudes entre los dos lenguajes tienen la intención de dar a los desarrolladores una idea de lo que JP QL puede hacer, pero al estar orientado a objetos, JP QL requiere una manera distinta de pensar. Si JP QL no es SQL, ¿qué es? Simplificando, JP QL es un lenguaje para hacer consultas sobre entidades. En lugar de tablas y columnas, este lenguaje trabaja con entidades y objetos. Proporciona una manera de expresar consultas en términos de entidades y sus relaciones, operando en el estado de persistencia de la entidad definido en el modelo objeto, no en el modelo físico de la base de datos.

Existen un par de razones importantes para considerar JP QL sobre SQL. Lo primero es la portabilidad. JP QL puede ser traducido a dialectos SQL en la mayoría de los proveedores de bases de datos. La segunda es que las consultas se dirigen al modelo de dominio de entidades persistentes, sin necesidad de saber exactamente cómo se asignan esas entidades a la base de datos.

Los ejemplos en este capítulo demuestran el poder que tienen hasta las expresiones más simples de JP QL. Adoptar JP QL no significa perder todos los recursos de SQL aprendidos. Una amplia selección de recursos SQL son ofrecidos directamente, incluyendo subconsultas, consultas agregadas, sentencia UPDATE y DELETE, un gran número de funciones SQL, y más.

4.1. Java Persistence Query Language

Antes de discutir sobre JP QL, es necesario revisar sus raíces. El *Enterprise JavaBeans Query Language* (EJB QL) se introdujo en la especificación EJB 2.0 para permitir a los desarrolladores a escribir buscadores portables y seleccionar métodos para los *beans* de entidades gestionados por contenedores.

Basado en un pequeño subconjunto de SQL, introdujo una manera de navegar a través de las relaciones de entidades para seleccionar datos y filtrar resultados. Desafortunadamente, estableció limitaciones estrictas en la estructura de la consulta, limitando los resultados tanto a las entidades como a los campos persistentes de la entidad. Las uniones internas (*inner joins*) entre las entidades eran posibles, pero usaban una notación singular. La versión inicial ni siquiera admitía ordenar los datos.

La especificación EJB 2.1 modificó el EJB QL un poco, permitiendo la ordenación de datos, e introdujo funciones básicas agregadas; pero, una vez más, la limitación de un único tipo de resultado dificultó el uso de agregados. Se podían filtrar los datos, pero no existía un equivalente a las expresiones de SQL GROUP BY y HAVING.

JP QL extiende significativamente EJB QL, eliminando muchas de las debilidades de las versiones previas y preservando la compatibilidad en retrospectiva. La siguiente lista describe algunos de los recursos disponibles más allá de EJB QL:

- Tipos de resultados únicos y múltiples
- Funciones agregadas, con cláusulas de ordenado y agrupado
- Una sintaxis de unión más natural, incluyendo soporte para uniones internas y externas
- Expresiones condicionales que incluyen subconsultas
- Consultas UPDATE y DELETE para realizar cambios de datos masivos
- Proyección de resultados en clases no persistentes

En las siguientes secciones se van a presentar en detalle las cláusulas y sentencias más utilizadas para la realización de consultas en JP QL, comparándolas con sus símiles en SQL para facilitar la comprensión de este lenguaje. Además, más adelante se explicará cómo definir consultas en JP QL, así como la manera de ejecutarlas, y algunos consejos de buenas prácticas para evitar posibles problemas a la hora de trabajar con las consultas.

4.2. Consultas SELECT

Las consultas SELECT son el tipo de consulta más significativo ya que permiten una recuperación masiva de datos. La estructura general de una SELECT es la siguiente:

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[ORDER BY <order_by_clause>]
```

Una consulta SELECT, tiene dos partes obligatorias: la cláusula SELECT que define el formato de los resultados de consulta y la cláusula FROM que apunta a la entidad o entidades de las que se quieren los datos. En este ejemplo se escribe una consulta SELECT que devuelve todos los empleados:

Ejemplo 1. Consulta SELECT que devuelve todos los empleados de la tabla EMPLOYEES

```
SELECT e
FROM Employee e
```

La estructura de esta consulta es muy similar a SQL, con la diferencia que la cláusula FROM hace referencia a entidades, no tablas. Además, a diferencia de las consultas de SQL, donde el alias es opcional, en JP QL es obligatorio.

La segunda diferencia, es que se utiliza el alias de la entidad para poder seleccionar todos los campos de esta.

4.2.1. Cláusula SELECT

Expresiones de ruta

Las expresiones de ruta son la base de las consultas. Se utilizan para navegar desde una entidad, ya sea a través de una relación con otra entidad o con una de las propiedades persistentes de la entidad.

El operador punto(.) significa navegación de ruta en una expresión. De esta manera, si la entidad Empleado se le ha asignado la variable de identificación `e`, `e.name`, es la expresión que nos devolvería el nombre del empleado.

Lo que hace que las expresiones de ruta sean tan poderosas es que no se limitan a una sola navegación. De esta manera, se puede construir expresiones de ruta como `e.department.name`, que es el nombre del departamento al que pertenece el empleado.

Entidades y Objetos

La estructura más simple de la cláusula SELECT es una única variable de identificación. Lo que devuelve la entidad completa a la cual está asociada esta variable. Por ejemplo, la consulta siguiente devolvería todos los departamentos de la empresa:

Ejemplo 2. Consulta SELECT que devuelve todos los departamentos de la tabla DEPARTMENTS

```
SELECT d
FROM Department d
```

La palabra clave OBJECT puede ser usada para resaltar, visualmente, que lo que se está devolviendo es un objeto entero. Es decir la expresión OBJECT(`e.department`) no sería válida ya que es una propiedad del objeto empleado, no el objeto en cuestión. A efectos prácticos la consulta anterior y la siguiente devuelven el mismo resultado:

Ejemplo 3. Consulta SELECT utilizando OBJECT que devuelve todos los departamentos de la tabla DEPARTMENTS

```
SELECT OBJECT(d)
FROM Department d
```

También, es posible utilizar la cláusula SELECT para devolver un único campo de una entidad. La siguiente consulta devuelve los nombres de todos los empleados:

Ejemplo 4. Consulta SELECT que devuelve el nombre de todos los empleados de la tabla EMPLOYEES

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

El resultado de esta consulta es un conjunto de *String*, por lo que la ejecución de esta consulta usando `getResultList()` devolverá una colección de cero o más *String*. La ejecución de las consultas se tratará más adelante en este capítulo.

La cláusula SELECT no solo devuelve propiedades de una única entidad, sino que también es posible devolver una entidad diferente por medio de una navegación por ruta. La siguiente consulta muestra cómo devolver una entidad diferente como resultado de una navegación por ruta:

Ejemplo 5. Navegación por ruta para devolver una entidad diferente

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

El resultado de esta consulta es el nombre del departamento al que pertenece cada empleado, ya que es el resultado de atravesar la relación del departamento entre Empleado y Departamento.

Para evitar valores duplicados, se añade el operador DISTINCT.

Ejemplo 6. Navegación por ruta para devolver una entidad diferente evitando repeticiones

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

El operador DISTINCT funciona prácticamente igual que el de SQL. Una vez se ha recogido el conjunto de resultados de una consulta, se eliminan los valores duplicados.

También es posible seleccionar objetos a los que se ha navegado en una expresión de ruta. La siguiente consulta devuelve los objetos `ContactInfo` para todos los empleados:

Ejemplo 7. Navegación por ruta para devolver un conjunto de objetos

```
SELECT e.contactInfo
FROM Employee e
```

Lo único que hay que tener en cuenta al seleccionar los objetos es que estos no se gestionarán. Si se realiza una consulta para devolver a los empleados y, a continuación, desde los resultados, se navega hasta los objetos `ContactInfo`, se obtendrán todos los que se hayan gestionado. La modificación de cualquiera de estos objetos se grabará una vez se confirme la operación.

Combinando Expresiones

Se pueden especificar múltiples campos en la misma cláusula SELECT separándolas con comas. El tipo de resultado de la consulta en este caso es una combinación de tipo objeto, cuyos elementos son los campos en el orden en que aparecieron en la consulta.

Considere la siguiente consulta que sólo devuelve el nombre y el salario de un empleado:

Ejemplo 8. Consulta SELECT con retorno más de un objeto

```
SELECT e.name, e.salary
FROM Employee e
```

Cuando esto se ejecuta, se devuelve una colección de cero o más *arrays* del tipo *Object*. Cada *array* en este ejemplo tiene dos elementos, el primero es un *String* que contiene el nombre del empleado y el segundo es un *Double* que contiene el salario del empleado.

Este tipo de consultas suelen hacerse para ahorrar un esfuerzo adicional a la hora de construir una instancia de la entidad. Por ello, cuando solo se quiere devolver un grupo pequeño de datos, es más útil utilizar este tipo de consultas que las que devuelven la entidad completa.

Expresiones Constructoras

Una forma más potente de la cláusula SELECT es almacenar los datos devueltos por la consulta en un tipo de objeto especificado por el usuario.

Ejemplo 9. Expresión constructora

```
SELECT NEW example.EmployeeDetails(e.name, e.salary, e.department.name)
FROM Employee e
```

El resultado de esta consulta es una clase de Java (`example.EmployeeDetails`). Como el procesador de consultas itera sobre los resultados de la consulta, utilizará un constructor que coincida con los tipos de expresión listados en la sección consulta. En este caso, los tipos de expresión son `String`, `Double`, `String`, por los que la consulta buscará un constructor con esos tipos para los argumentos.

Se debe hacer referencia al tipo de objeto de resultado empleando el nombre literal de la clase. Sin embargo, la clase no tiene que ser asignada a la base de datos. Cualquier constructor compatible con las expresiones listadas en la cláusula SELECT puede ser usado en una expresión constructora.

4.3. Cláusula FROM

La cláusula FROM se utiliza para declarar una o más variables de identificación, que pueden o no ser resultado de un JOIN, donde la consulta podrá mostrar sus resultados. La sintaxis de la cláusula FROM consiste en una o más variables de identificación y declaraciones de cláusulas de JOIN.

4.3.1. Variables de identificación

La variable de identificación es el punto de partida de todas las consultas. Cada consulta debe tener al menos una variable de identificación definida en la cláusula FROM, y esa variable debe corresponder a un tipo de entidad. Cuando al declarar variables de identificación no se utiliza una expresión de ruta, se denomina declaración de variables de rango.

Las declaraciones de variables de rango utilizan la sintaxis `<nombre_de_entidad>[AS] <identificador>`. Se ha estado usando esta sintaxis en ejemplos anteriores, pero sin la necesidad de utilizar la palabra clave AS. El identificador debe seguir las reglas de nomenclatura estándar de Java y puede ser referenciado a lo largo de la consulta de forma insensible a mayúsculas y minúsculas. Se pueden especificar varias declaraciones separándolas con comas.

Las expresiones de ruta también se pueden aliar a variables de identificación en el caso de uniones y subconsultas.

4.4. JOINS

Un JOIN es una consulta que combina resultados de múltiples entidades. Un JOIN en JP QL funciona de manera parecida a cómo lo hace en SQL. Una vez que la consulta se traduce a SQL, es muy probable que las uniones entre entidades produzcan uniones similares entre las tablas equivalentes a esas entidades mapeadas. Por lo tanto, es importante comprender cuándo se producen las uniones para escribir consultas eficaces.

Las uniones se producen cuando se cumple alguna de las siguientes condiciones en una consulta SELECT:

- En la cláusula FROM se enumeran dos o más declaraciones de variables de rango y aparecen en la cláusula SELECT.
- El operador JOIN se utiliza para ampliar una variable de identificación utilizando una expresión de ruta.
- Una expresión de ruta en cualquier parte de la consulta navega a través de un campo de asociación, a la misma o hacia una entidad diferente.
- Una o más condiciones WHERE comparan atributos de diferentes variables de identificación.

Un INNER JOIN entre dos entidades devuelve los objetos de ambos tipos de entidades que cumplen todas las condiciones del JOIN. La ruta de navegación de una entidad a otra es una forma de INNER JOIN.

El OUTER JOIN de dos entidades es la unión de objetos de ambos tipos de entidades que satisfacen las condiciones del JOIN junto al conjunto de objetos de un tipo de entidad que no tienen ninguna condición JOIN en la otra.

En ausencia de condiciones de JOIN entre dos entidades, las consultas producirán un producto cartesiano entre las dos entidades. Los productos cartesianos son raros en las consultas JP QL dadas las capacidades de navegación del idioma, pero son posibles si se especifican dos declaraciones de variables de rango en la cláusula FROM sin condiciones adicionales especificadas en la cláusula WHERE.

A continuación, se verán con más detalle cada tipo de JOIN.

4.4.1. INNER JOIN (Unión interna)

Todas las consultas anteriores han utilizado la cláusula FROM poniéndole un alias a la variable de identificación. Sin embargo, como lenguaje relacional, JP QL soporta consultas que se basan en múltiples entidades y las relaciones entre ellas.

La forma típica de las uniones internas entre dos entidades es el uso del operador JOIN en la cláusula FROM.

Campos del operador JOIN y Asociación de recogida.

La sintaxis del INNER JOIN utilizando el operador JOIN es:

```
[INNER] JOIN <path_expression>[AS] <identificador>
```

Considerando el siguiente ejemplo:

Ejemplo 10. Consulta con una cláusula JOIN

```
SELECT p
FROM Employee e JOIN e.phones p
```

Esta consulta utiliza el operador JOIN para unir la entidad empleado con la entidad teléfono. La condición del JOIN en esta consulta está definida por el ORM de la relación entre los teléfonos. No es necesario especificar criterios adicionales para vincular a las dos entidades. Al unir las dos entidades, esta consulta devuelve todas las instancias de entidades teléfono asociadas con los empleados de la empresa.

La sintaxis de la unión es similar a la de las expresiones JOIN soportadas por SQL. También es posible realizar la misma consulta en la forma tradicional del JOIN:

Ejemplo 11. Consulta JOIN sin usar la cláusula JOIN

```
SELECT p.id, p.phone_num, p.type, p.emp_id
FROM emp e, phone p
WHERE e.id = p.emp_id
```

El mapeo de la tabla Teléfono reemplaza la expresión `e.phones`. La cláusula WHERE también incluye los criterios necesarios para unir las dos tablas a través de las columnas unidas definidas por el mapeo de teléfonos.

El uso de JOIN permite evitar que Java tenga que acceder a los campos de una entidad por medio de otra. De tal manera que, mediante la unión de las dos tablas, se puede acceder a todos los campos de la entidad Teléfono simplemente navegando por la tabla de unión.

Ejemplo 12. Navegación usando la cláusula JOIN

```
SELECT p.number
FROM Employee e JOIN e.phones p
```

Para evitar que una expresión de ruta no pueda continuar desde un campo o una colección, el campo de asociación de la colección debe estar unido en la cláusula FROM de forma que se cree una nueva variable de identificación para la ruta.

4.4.2. IN versus JOIN

La consulta vista anteriormente, se puede expresar mediante la cláusula IN, devolviendo el mismo resultado

Ejemplo 13. Cláusula IN para la navegación

```
SELECT DISTINCT p
FROM Employee e, IN(e.phones) p
```

El operador IN tiene por objeto indicar que la variable p es una enumeración de la colección de teléfonos. En cambio, el operador JOIN es una forma más poderosa y expresiva de declarar relaciones, además de ser el operador recomendado para las consultas.

4.4.3. Operador JOIN y campos de Asociación de valor individual

El operador JOIN trabaja tanto con expresiones de ruta de asociación de valor de colección como con expresiones de ruta de asociación de valor único

Ejemplo 14. Cláusula JOIN con expresión de ruta de asociación de valor de colección

```
SELECT d
FROM Employee e JOIN e.department d
```

Esta consulta define una unión de Empleado a Departamento a través de la relación de departamento. Esto es semánticamente equivalente a usar una expresión de ruta en la cláusula SELECT para obtener el departamento para el empleado.

La posibilidad de INNER JOINS implícitos resultantes de las expresiones de ruta es algo de lo que hay que ser consciente. Considere el siguiente ejemplo que devuelve los distintos departamentos situados en California que están participando en el proyecto Release1:

Ejemplo 15. Consulta para obtener todos los empleados que participan en un Proyecto fijado y viven en California

```
SELECT DISTINCT e.department
FROM Project p JOIN p.employees e
WHERE p.name = 'Release1' AND e.address.state = 'CA'
```

En realidad, aquí hay cuatro uniones lógicas, no dos. El traductor tratará la consulta como si hubiera sido escrita con uniones explícitas entre las distintas entidades.

Ejemplo 16. Cláusula SQL equivalente a los dos ejemplos anteriores

```
SELECT DISTINCT d.id, d.name
FROM project p, emp_projects ep, emp e, dept d, address a
WHERE p.id = ep.project_id AND
      ep.emp_id = e.id AND
      e.dept_id = d.id AND
      e.address_id = a.id AND
      p.name = 'Release1' AND
      a.state = 'CA'
```

4.4.4. Condiciones JOIN en la cláusula WHERE

Las consultas de SQL han unido, tradicionalmente, las tablas mediante la enumeración de las tablas que deben unirse en la cláusula FROM y los filtros necesarios en la cláusula WHERE.

El ejemplo 14 podría haberse escrito así.

Ejemplo 17. Consulta usando la cláusula WHERE

```
SELECT DISTINCT d
FROM Department d, Employee e
WHERE d = e.department
```

Este estilo de consulta se utiliza normalmente para compensar la falta de una relación explícita entre dos entidades en el modelo de dominio.

Ejemplo 18. Consulta utilizando la expression IS NOT EMPTY

```
SELECT d, m
FROM Department d, Employee m
WHERE d = m.department AND
m.directs IS NOT EMPTY
```

En el ejemplo anterior, se utiliza la expresión IS NOT EMPTY que verifica que la colección *directs* no está vacía.

4.4.5. Uniones múltiples

En algunos casos es necesario unir en cascada. Por ejemplo, la consulta siguiente devuelve el conjunto de proyectos de un empleado que pertenece a un departamento.

Ejemplo 19. Consulta con múltiples uniones

```
SELECT DISTINCT d
FROM Project p JOIN p.employees e JOIN e.department d
JOIN e.address a
WHERE p.name = 'Release1' AND a.state = 'CA'
```

El procesador de consultas interpresa la cláusula FROM de izquierda a derecha, de tal manera, que es posible hacer referencia a una variable declara anteriormente en otras expresiones JOIN.

4.4.6. MAP JOINS (mapa de Uniones)

Una expresión de ruta que navega a través de una asociación de valor de colección implementado como un Mapa es un caso especial. A diferencia de una colección normal, cada elemento de un mapa corresponde con dos piezas de información: la clave y el valor. Cuando se trabaja con JP QL es importante señalar que las variables de identificación basadas en mapas se refieren al valor por defecto. Por ejemplo, considérese el caso en el que la relación telefónica del Empleado se modela como un mapa, donde la clave es el tipo de número y el valor es el número de teléfono.

Ejemplo 20. Consulta usando MAPS con y sin VALUE

```
SELECT e.name, p
FROM Employee e JOIN e.phones p
```

Este comportamiento puede ser resaltado explícitamente mediante el uso de la palabra clave VALUE.

```
SELECT e.name, VALUE(p)
FROM Employee e JOIN e.phones p
```

Para acceder a la clave en lugar del valor de un elemento de mapa determinado, se puede utilizar la palabra clave KEY para anular el comportamiento predeterminado y devolver el valor de la clave para un elemento de mapa determinado. El siguiente ejemplo muestra cómo añadir el tipo de teléfono a la consulta anterior:

Ejemplo 21. Consulta especificando la KEY y el VALUE

```
SELECT e.name, KEY(p), VALUE(p)
FROM Employee e JOIN e.phones p
WHERE KEY(p) IN ('Work', 'Cell')
```

Hay que tener en cuenta que, en cada uno de los ejemplos de unión de mapas, se une una entidad contra uno de sus atributos de tipo mapa y se obtiene un par clave, valor o valor clave (*entry*). Sin embargo, cuando se ve desde la perspectiva de las tablas, la unión sólo se hace a nivel de la clave primaria de la entidad de origen y de los valores del mapa. Actualmente no hay ninguna herramienta disponible en JPA para unirse a la entidad fuente y a las claves del Mapa.

4.4.7. OUTER JOIN (Unión externa)

Un OUTER JOIN entre dos entidades produce un dominio en el que sólo se requiere que un lado de la relación esté completo. Es decir, la unión externa de `Empleado a Departamento` mediante el campo departamento de empleado, devuelve a todos los empleados y el departamento al cual estén asignados, exista o no ese departamento, pero solo si el departamento tiene asignado un empleado este será devuelto.

Se especifica una unión externa utilizando la siguiente sintaxis:

```
LEFT[OUTER] JOIN <path_path expresión>[AS]<identificador>.
```

La siguiente consulta muestra la unión externa entre dos entidades:

Ejemplo 22. Consulta usando la cláusula LEFT JOIN

```
SELECT e, d
FROM Employee e LEFT JOIN e.department d
```

Si el empleado no ha sido asignado a un departamento, el objeto departamento será nulo.

En sintaxis propia de SQL sería:

Ejemplo 23. Consulta especificando la KEY y el VALUE

```
SELECT e.id, e.name, e.salary, e.manager_id, e.dept_id, e.address_id,
       d.id, d.name
FROM employee e LEFT OUTER JOIN department d
ON (d.id = e.department_id)
```

El SQL resultante muestra que cuando se genera una unión externa a partir de JP QL siempre especifica una condición ON de igualdad entre la columna de unión que mapea la relación a través de la cual se está uniendo y la clave primaria a la que se está haciendo referencia.

Se puede usar la expresión ON para añadir restricciones a los objetos que se han devuelto desde el lado derecho de la unión.

Ejemplo 24. Consulta con la cláusula ON

Se puede añadir una cláusula ON al JP QL del ejemplo 15 de tal manera que los departamentos devueltos sean sólo aquellos que empiezan con la cadena de texto `QA'.

```
SELECT e, d
FROM Employee e LEFT JOIN e.department d
ON d.name LIKE 'QA%'
```

Esta consulta todavía devuelve a todos los empleados, pero los resultados no incluirán ningún departamento que no coincida con la condición ON añadida. El SQL generado tendría el siguiente aspecto esto:

Ejemplo 25. Equivalente en SQL

```
SELECT e.id, e.name, e.salary, e.department_id, e.manager_id,
```

```
            e.address_id, d.id, d.name
FROM employee e left outer join department d
ON ((d.id = e.department_id) and (d.name like 'QA%'))
```

Destacar que esta consulta es muy diferente de usar una expresión WHERE ya que la cláusula WHERE provoca un INNER JOIN entre Empleado y Departamento, por lo que esta consulta sólo devolvería a los empleados que estaban en un departamento con un nombre prefijado 'QA'.

4.4.8. FECTH JOINS

Los FETCH JOIN están pensados para ayudar a los programadores a optimizar el acceso a la base de datos y a preparar los resultados de las consultas. Permiten que en las consultas se especifique una o más relaciones que deben ser navegadas y preprogramadas por el motor de consultas para que no tarden en cargar en tiempo de ejecución.

Para implementar FETCH JOIN, el proveedor necesita convertir la asociación de búsqueda en una unión regular del tipo apropiado: interna por defecto, o externa si se especificó la palabra clave LEFT. La expresión SELECT de la consulta también necesita ser expandida para incluir la relación JOIN. Expresado en JP QL, el FETCH JOIN se expresa:

Ejemplo 26. Consulta usando la cláusula FETCH JOIN

```
SELECT e, a
FROM Employee e JOIN e.address a
```

La única diferencia es que el proveedor no devuelve las entidades de dirección a la persona que llama. Debido a que los resultados se procesan a partir de esta consulta, el motor de consulta crea la entidad de dirección en memoria y la mapea la entidad empleado, pero luego la elimina de la colección de resultados que crea para el cliente. Esto aclara la relación de direcciones a la que se puede acceder navegando desde la entidad Empleado.

Una consecuencia de la implementación de los FETCH JOIN de esta manera es que la obtención de una asociación de colección da como resultado resultados duplicados. Por ejemplo, considérese una consulta del departamento donde hay una gran relación de los empleados de la entidad del Departamento. La consulta FETCH JOIN, esta vez utilizando un enlace externo para asegurar que se recuperan los departamentos sin empleados, se escribiría de la siguiente manera:

Ejemplo 27. Consulta usando la cláusula LEFT FETCH JOIN

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
```

En cambio, si se realiza una unión externa entre las dos tablas, se pueden ver todos los empleados que pertenecen a cada departamento.

Ejemplo 28. Unión externa entre dos tablas con LEFT JOIN

```
SELECT d, e
FROM Department d LEFT JOIN d.employees e
```

Cada entidad del Departamento tiene ahora una colección de empleados totalmente completa, pero el cliente recibiría una referencia a cada departamento por empleado. Es decir, si se recuperaran cuatro departamentos con cinco empleados cada uno, el resultado sería una colección de 20 instancia del Departamento, con cada departamento duplicado cinco veces. Para evitar esta repetición los valores duplicados se debe utilizar el operador DISTINCT.

Dados los resultados anteriores, realizar una unión de una colección a otra puede que no sea la forma más apropiada de cargar entidades relacionadas en todos los casos. si una colección requiere que se busque de forma regular, se debe considerar dicha posibilidad. Algunos proveedores de persistencia también ofrecen lecturas de lotes como una alternativa para obtener uniones que emiten múltiples consultas en un solo lote y luego correlacionan los resultados con relaciones de dependencia. Otra alternativa es utilizar un gráfico de entidad para determinar dinámicamente los atributos de la relación que se van a cargar mediante una consulta.

4.5. Cláusula WHERE

WHERE permite especificar condiciones para limitar los valores devueltos en una consulta. Prácticamente todo lo que visto para esta cláusula se puede aplicar a **HAVING**.

Esta cláusula funciona de manera sencilla. Simplemente limita la consulta escribiendo WHERE y después la condición.

4.5.1. Parámetros de entrada

Los parámetros que se usen para las condiciones pueden ser tanto su nombre o su posición.

- Mediante el nombre. Se escribe "`:nombre_del_parámetro`" en la consulta. Distingue entre mayúsculas y minúsculas. Véase a continuación:

Ejemplo 29. Cláusula WHERE mediante nombre

```
SELECT e
FROM Employee e
WHERE e.salary > :sal
```

Devuelve aquellos salarios de los empleados en la tabla Empleado que sean mayores del valor del registro "sal"

- Mediante su posición. Para ello se escribe el número de la variable precedido por un signo de interrogación tal que: "?1", "?2", etc. Véase un ejemplo análogo:

Ejemplo 30. Cláusula WHERE mediante posición

```
SELECT e
FROM Employee e
WHERE e.salary > ?1
```

4.5.2. Operadores

La forma de expresar en JP QL las condiciones proviene en gran parte del lenguaje SQL. La diferencia clave entre las expresiones condicionales en JP QL y SQL, es que JP QL, al ser dinámico, permite identificar y acceder a la dirección de memoria de las distintas variables mientras se evalúa una expresión.

Los operadores son también similares a los que se usan en la cláusula de WHERE en SQL.

- Operador de navegación (.)
- Aritméticos (+, -, *, /)
- Lógicos (AND, OR, NOT)
- Comparación (=, >, <, >=, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF])

4.5.3. Expresiones BETWEEN

Se utilizan para delimitar el rango de valores en el que se desea que encuentre un resultado. Para ello se pueden utilizar valores numéricos, strings y fechas. Véase el siguiente ejemplo.

Ejemplo 31. Expresión BETWEEN

```
SELECT e
FROM Employee e
WHERE e.salary BETWEEN 40000 AND 45000
```

El resultado de esta consulta serán aquellos empleados cuyo salario esté en ese rango incluyendo 40000 y 45000. Una forma análoga con operadores de comparación básicos sería:

Ejemplo 32. Equivalente a expresión BETWEEN

```
SELECT e
FROM Employee e
WHERE e.salary >= 40000 AND e.salary <= 45000
```

El operador BETWEEN también puede ser negado con el operador NOT.

4.5.4. Expresiones LIKE

Sirven para comparar textos (*string*). Tienen dos operadores que se escriben dentro del *string*: el operador (_) que equivale a cualquier carácter y el (%), que equivale a cualquier cadena de texto. Véase el ejemplo:

Ejemplo 33. Expresión LIKE

```
SELECT d
FROM Departament d
WHERE d.name LIKE '__Ent%'
```

El resultado de esta consulta serán aquellos departamentos que tengan dos caracteres al principio (nótese que hay dos _ unidas), luego contengan el *string* 'Ent' y a continuación tengan cualquier cadena de texto. Un ejemplo sería iVEntas o SrEntas, pero no COmpras.

4.5.5. Subconsultas

Pueden ser usadas combinándolas con WHERE o HAVING. Una subconsulta es una consulta SELECT que irá entre paréntesis dentro de una condición. En el siguiente ejemplo:

Ejemplo 34. Subconsulta

```
SELECT e
FROM Employee e
WHERE e. salary = (SELECT MAX(emp.salary)
                   FROM Employee emp)
```

El resultado será el empleado con el mayor salario entre todos los empleados.

La posibilidad de referirse a una variable de la consulta principal en una subconsulta, permite a las dos estar relacionadas, como en el siguiente ejemplo:

Ejemplo 35. Cláusula WHERE mediante nombre

```
SELECT e
FROM Employee e
WHERE EXISTS (SELECT p
                FROM Phone p
               WHERE p.employee = e AND p.type = 'Cell')
```

El resultado será aquellos empleados que tengan un número de teléfono móvil. Más adelante se verá la palabra reservada EXISTS.

4.5.6. Expresiones IN

Permite saber si un valor pertenece a una lista de textos o números. El siguiente ejemplo mostrará aquellos empleados que viven en Nueva York o California:

Ejemplo 36. Expresión IN

```
SELECT e
FROM Employee e
WHERE e.address.state IN ('NY', 'CA')
```

Estas expresiones pueden utilizar subconsultas a su vez, donde la lista será el resultado de esa subconsulta. El siguiente ejemplo devuelve aquellos empleados que trabajan en departamentos que contribuyen con proyectos cuyo nombre empieza por QA.

Ejemplo 37. Expresión IN con subconsulta

```
SELECT e
FROM Employee e
WHERE e.departament IN (SELECT DISTINCT d
                            FROM Departament d JOIN d.employee de JOIN
                            de.projects p
                            WHERE p.name LIKE 'QA%')
```

La expresión IN puede ser negada con el operador NOT. En este ejemplo se seleccionan aquellos números de teléfono que no son ni de la oficina ni de casa.

Ejemplo 38. Expresión IN negada con NOT

```
SELECT p
FROM Phone p
```

```
WHERE p.type NOT IN ('Office', 'Home')
```

4.5.7. Expresiones de colección

El operador IS EMPTY es el equivalente de IS NULL, pero para colecciones. Puede ser negado también (IS NOT EMPTY) para ver que una colección contenga al menos un valor.

El siguiente ejemplo mostrará los empleados que sean directores de al menos otro empleado:

Ejemplo 39. Expresión IS NOT EMPTY

```
SELECT e
FROM Employee e
WHERE e.directs IS NOT EMPTY
```

Para traducir IS EMPTY a SQL han de utilizarse subconsultas. Un ejemplo equivalente al anterior sería el siguiente:

Ejemplo 40. Equivalente a IS NOT EMPTY con subconsulta

```
SELECT m
FROM Employee m
WHERE (SELECT COUNT(e)
        FROM Employee e
        WHERE e.manager = m) > 0
```

El operador MEMBER OF muestra si un valor es miembro de una colección. También puede ser negado (NOT MEMBER OF). La siguiente consulta mostrará aquellos empleados que están asignados a un determinado proyecto.

Ejemplo 41. Operador MEMBER OF

```
SELECT e
FROM Employee e
WHERE :project MEMBER OF e.projects
```

También en este caso para traducirlo a SQL sería necesario utilizar subconsultas. Un ejemplo equivalente al anterior sería el siguiente:

Ejemplo 42. Equivalente a MEMBER OF con subconsulta

```
SELECT e
```

```
FROM Employee e
WHERE :project IN (SELECT p
                   FROM e.projects p)
```

4.5.8. Expresión EXISTS

La condición EXISTS devuelve verdadero si una subconsulta devuelve cualquier línea. El operador EXISTS también puede ser negado con el operador NOT. La siguiente consulta selecciona a todos los empleados que no tienen un teléfono móvil:

Ejemplo 43. Expresión EXISTS

```
SELECT e
FROM Employee e
WHERE NOT EXISTS (SELECT p
                  FROM e.phones p
                  WHERE p.type = 'Cell')
```

4.5.9. Expresiones ANY, ALL y SOME

Se usan junto con los comparadores básicos: =, >, <, >=, <= y <>. ALL devuelve *true* cuando todos los elementos de la lista cumplen la condición, va comparando la condición a la izquierda con todos los elementos de la derecha. Para que la condición sea efectiva, todos los elementos de la lista deben cumplir esa condición.

Ejemplo 44. Expresión ALL

```
SELECT e
FROM Employee e
WHERE e.directs IS NOT EMPTY AND
      e.salary < ALL(SELECT d.salary
                     FROM e.directs d)
```

Esta consulta devuelve aquellos jefes cuyo salario es menor que todos los empleados que trabajan para ellos.

ANY trabaja de manera análoga, pero si encuentra al menos un elemento de la lista que cumple la condición ya devuelve *true*. Tiene un funcionamiento muy parecido al de las expresiones IN. Usando el mismo ejemplo que en las expresiones IN, pero con ANY:

Ejemplo 45. Expresión ANY

```
SELECT e
FROM Employee e
WHERE e.departament = ANY (SELECT DISTINCT d
                           FROM Departament d JOIN d.employee de JOIN
                           de.projects p
                           WHERE p.name LIKE 'QA%')
```

4.6. Herencia y polimorfismo

JPA también trabaja con herencia entre entidades, por lo que varias subclases de una entidad pueden ser el resultado de una misma consulta.

Por ejemplo, `Project` es la clase madre para `QualityProject` y `DesignProject`. Si se utiliza una variable de identificación directamente de la entidad `Project`, el resultado de la consulta incluirá una mezcla de objetos Java tanto de `Project` como de `QualityProject` y `DesignProject`, y con el resultado se puede hacer *cast* según sea necesario.

4.6.1. Selección entre subclases

Si se quiere restringir el resultado de una consulta a una subclase particular, después de la cláusula FROM puede utilizarse esa subclase en vez de la entidad raíz. Sin embargo, si el resultado quiere ampliarse a más de una subclase pero no a todas, la cláusula WHERE unida a TYPE puede incluir las subclases que se quieran.

En este ejemplo sólo se devolverán los proyectos de calidad y de diseño:

Ejemplo 46. Selección de subclases con WHERE TYPE

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

Nótese que no se han puesto `'DesignProject'` ni `'QualityProject'` ya que JP QL toma los nombres de las subclases como entidades no como strings. Estas consultas también pueden hacerse dinámicas, y que tome el valor de esa subclase dado por el programador:

Ejemplo 47. Selección dinámica de subclases

```
SELECT p
FROM Project p
WHERE TYPE(p) = :projectType
```

4.6.2. *Downcasting*

Puede accederse a un atributo de una subclase, pero cuando la consulta trata con la clase madre, hay que usar un *cast* al que se denomina *downcasting.* Esta técnica consiste en hacer que una consulta que se refiere a una clase, sea aplicada a una subclase de esta específica, usando el operador TREAT.

En este ejemplo, el resultado serán todos los proyectos de diseño y los de calidad con un índice de calidad mayor de 4.

Ejemplo 48. Downcasting con WHERE TREAT

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

La sintaxis de la expresión empieza con TREAT seguido de su argumento entre paréntesis. El argumento será el alias de la clase madre, la palabra reservada AS y el nombre de la subclase.

4.7. **Expresiones escalares**

Una expresión escalar es un valor literal, una secuencia aritmética, una función, una expresión *type* o una expresión *case* que tiene como resultado un valor escalar, es decir, un número. Pueden ser usadas en un SELECT o como condición en una expresión WHERE o HAVING.

4.7.1. Valores literales

En JP QL pueden usarse strings, valores numéricos, *booleans*, *enums*, entidades y fechas.

En el siguiente ejemplo se usa un *enum* llamado *PhoneType*.

Ejemplo 49. Uso de un *enum* en una expresión condicional

```
SELECT e
FROM Employee e JOIN e.phoneNumbers p
WHERE KEY(p) = com.acme.PhoneType.Home
```

Para fechas, se utiliza el mismo formato que JBDC, tal que:

```
{d 'yyyy-mm-dd'}                    e.g. {d '2009-11-05'}
{t 'hh-mm-ss'}                             e.g. {t '12-45-52'}
{ts 'yyyy-mm-dd hh-mm-ss.f'}        e.g. {ts '2009-11-05 12-45-52.325'}
```

4.7.2. Funciones

La tabla a continuación resume la sintaxis para las distintas funciones soportadas por JPA.

Función	**Devuelve**
ABS (número)	El valor absoluto del número en el argumento con el mismo tipo (*int, float* o *double*)
CONCAT(*string1*, *string2*)	Un nuevo string que es la concatenación de los dos en el argumento
CURRENT_DATE	La fecha actual definida en la base de datos del servidor
CURRENT_TIME	La hora actual definida en la base de datos del servidor
CURRENT_TIMESTAMP	El *timestamp* actual definido en la base de datos del servidor
INDEX(alias)	La posición de una entidad en una lista ordenada
LENGTH(string)	El número de caracteres en un string
LOCATE(*string1*, *string2* [,start])	La posición del string*1* en el string*2,* pudiendo empezar por la posición indicada por *start*. El resultado es 0 si el string no se encuentra
LOWER(string)	El string del argumento en minúsculas
MOD(num1, num2)	El módulo del cociente entre num1 y num2, como un *integer*
SIZE(colección)	El número de elementos de un colección, devolviendo 0 si está vacía
SQRT(número)	La raíz cuadrada del número en el argumento como un *double*
SUBSTRING(string, start, end)	Una parte del string*,* empezando en la posición indicada en starthasta la posición *end*. La posición de un string empieza en 1
UPPER(string)	El string del argumento en mayúsculas
TRIM([[LEADING|TRAILING|BOTH] [*char*] FROM] *string)*	Elimina los caracteres LEADING y/o TRAILING de un string*.* Si no se usa TREADING, LEADING o BOTH, ambos caracteres se eliminan. El carácter por defecto de TRIM son los espacios

La función SIZE requiere especial atención ya que es una notación abreviada para una subconsulta agregada. A continuación, se muestra un ejemplo en el que se devuelven aquellos departamentos con sólo dos empleados:

Ejemplo 50. Función SIZE

```
SELECT d
FROM Department d
WHERE SIZE(d.employees) = 2
```

Utilizando subconsultas se haría de la siguiente forma:

Ejemplo 51. Equivalente a la función SIZE con subconsultas

```
SELECT d
FROM Department d
WHERE (SELECT COUNT(e)
        FROM d.employees e) = 2
```

Cuando se utilizan colecciones ordenadas, cada elemento de la misma contiene dos datos diferentes: el valor almacenado en la colección y su posición numérica dentro de esta. Las consultas pueden utilizar la función INDEX para determinar la posición numérica de cualquier dato en una colección. Por ejemplo, si los números de teléfono de un empleado se guardan en orden de prioridad, el siguiente código devolvería el primer (y más importante) número para cada empleado:

Ejemplo 52. Función INDEX

```
SELECT e.name, p.number
FROM Employee e JOIN e.phones p
WHERE INDEX(p) = 0
```

4.7.3. Funciones nativas de base de datos

Las funciones de base de datos pueden ser usadas en JP QL usando la expresión FUNCTION. Esta palabra reservada, seguida por la función y su argumento, tendrá como resultado un número, un *boolean,* un *string* o una fecha.

La siguiente consulta llama a una función de base de datos llamada `shouldGetBonus`. Sus parámetros serán el departamento del empleado y el proyecto en el que esté trabajando. Este código fijará aquellos empleados que recibirán una prima.

Ejemplo 53. Expresión FUNCTION

```
SELECT DISTINCT e
FROM Employee e JOIN e.projects p
WHERE FUNCTION('shouldGetBonus', e.dept.id, p.id)
```

4.7.4. Expresiones CASE

Estas expresiones permiten introducir lógica condicional en una consulta. Se pueden expresar de cuatro maneras, dependiendo de la flexibilidad necesaria de la consulta. La primera y más flexible es del tipo:

```
CASE {WHEN <cond_expr> THEN <scalar_expr>}+ ELSE <scalar_expr> END
```

La expresión WHEN es necesaria en este caso (al menos una condición), para que sirva como patrón de búsqueda. Después de encontrar un caso válido, evalúa la expresión escalar (argumento de THEN), y la devuelve como resultado. Si ninguna de las condiciones del WHEN se cumplen, se evalúa y devuelve la expresión escalar que va con la palabra reservada ELSE.

En el siguiente ejemplo se enumera el nombre y tipo de proyecto de los empleados.

Ejemplo 54. Expresión CASE forma 1

```
SELECT p.name,
       CASE WHEN TYPE(p) = DesignProject THEN 'Development'
            WHEN TYPE(p) = QualityProject THEN 'QA'
            ELSE 'Non-Development'
       END
FROM Project p
WHERE p.employees IS NOT EMPTY
```

El uso de la expresión CASE se hace como parte de la cláusula SELECT.

Una variación de la expresión CASE general es la más simple. Se escribe tal que:

```
CASE <value> {WHEN <scalar_expr1> THEN <scalar_expr2>}+
ELSE <scalar_expr> END
```

En vez de comprobar una condición WHERE, identifica un valor y utiliza una expresión CASE en cada sentencia WHEN. El argumento `<value>`, puede ser un alias o una expresión TYPE. A continuación, se puede ver un ejemplo:

Ejemplo 55. Expresión CASE forma 2

```
SELECT p.name,
       CASE TYPE(p)
            WHEN DesignProject THEN 'Development'
            WHEN QualityProject THEN 'QA'
            ELSE 'Non-Development'
       END
FROM Project p
WHERE p.employees IS NOT EMPTY
```

La tercera forma de la expresión CASE es la expresión COALESCE. Esta acepta una secuencia de una o más expresiones escalares, de la siguiente forma:

```
COALESCE(<scalar_expr> {,<scalar_expr>}+)
```

Las expresiones escalares serán resueltas en orden. La primera en devolver un valor no nulo, será el resultado de la expresión. El siguiente ejemplo devolverá el nombre de cada departamento, o su id si algún departamento no se ha nombrado:

Ejemplo 56. Expresión CASE forma 3 (expresión COALESCE)

```
SELECT COALESCE(d.name, d.id)
FROM Department d
```

La última forma acepta dos expresiones escalares, resolviendo ambas. Si el resultado de estas es igual, el resultado de la expresión CASE es nulo. Si no, devuelve el resultado de la primera expresión. Para ello se utiliza la palabra NULLIF.

```
NULLIF(<scalar_expr1>, <scalar_expr2>)
```

En el siguiente ejemplo se devuelve la cuenta de todos los departamentos y de aquellos que no se llaman 'QA':

Ejemplo 57. Expresión CASE forma 4 (expresión NULLIF)

```
SELECT COUNT(*), COUNT(NULLIF(d.name, 'QA'))
FROM Department d
```

4.8. ORDER BY

Las consultas pueden ser ordenadas con la cláusula ORDER BY. Las palabras opcionales ASC y DESC, después de la cláusula ORRDER BY, se utilizan para ordenar de manera ascendente o descendente. Por defecto lo hará de manera ascendente.

El siguiente ejemplo demuestra como ordenar por un solo campo:

Ejemplo 58. Expresión ORDER BY para ordenar por un campo

```
SELECT e
FROM Employee e
ORDER BY e.name DESC
```

También se puede ordenar en función de más de un campo, donde lo hará en orden.

Ejemplo 59. Expresión ORDER BY para ordenar por dos campos

```
SELECT e, d
FROM Employee e JOIN e.department d
ORDER BY d.name, e.name DESC
```

Pueden utilizarse alias también como patrón para ordenar, y definidos en la misma consulta como en el siguiente ejemplo:

Ejemplo 60. Expresión ORDER BY utilizando alias

```
SELECT e.name, e.salary * 0.05 AS bonus, d.name AS deptName
FROM Employee e JOIN e.department d
ORDER BY deptName, bonus DESC
```

Pero los campos que se usen para ORDER BY, deben estar en la cláusula SELECT de la misma consulta. El siguiente ejemplo sería no válido:

Ejemplo 61. Expresión ORDER BY no válida

```
SELECT e.name
FROM Employee
ORDER BY e.salary DESC
```

4.9. Consultas agregadas

Estas son variaciones de las consultas normales. Una consulta se considerada agregada si usa una función agregada o contiene expresiones GROUP BY y/o HAVING. La sintaxis de una consulta agregada típica quedaría:

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[GROUP BY <group_by_clause>]
[HAVING <conditional_expression>]
[ORDER BY <order_by_clause>]
```

La ventaja de una consulta agregada viene de usar funciones agregadas en datos agrupados, como, por ejemplo:

Ejemplo 62. Consulta agregada

```
SELECT AVG(e.salary)
FROM Employee e
```

Este ejemplo devuelve el salario medio de todos los empleados en una compañía. Ahora con la siguiente variación del ejemplo anterior pueden agruparse los resultados:

Ejemplo 63. Consulta agregada con resultados agrupados

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

Esta consulta devuelve el nombre de cada departamento y el salario medio de los empleados en ese departamento. Esto podría incluso hacerse más específico para que el salario de los directores no sea incluido:

Ejemplo 64. Consulta agregada con resultados agrupados sin incluir el salario de los directores

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
WHERE e.directs IS EMPTY
GROUP BY d.name
```

Para poder manejar los datos a voluntad del programador se utilizan las funciones agregadas que dan mucha capacidad de agrupación y selección de datos.

4.9.1. Funciones agregadas

Son cinco: AVG, COUNT, MAX, MIN y SUM.

AVG

Toma el nombre o alias de un campo como argumento y calcula el valor medio de todos los registros que tenga en ese campo. Los registros de ese campo deben ser numéricos, y como resultado AVG dará un valor *Double.*

Count

Toma tanto un variable de identificación como un alias o nombre de un campo como argumento. El resultado es un valor *Long* que representa el número de registros que tiene un campo. El argumento puede estar precedido por la palabra DISTINCT para que solo cuente aquellos registros diferentes y descarte los que se repiten.

MAX

Devuelve el mayor valor entre todos los registros de un campo.

MIN

Devuelve el menor valor entre todos los registros de un campo.

SUM

Calcula la suma de todos los valores de un campo, teniendo como argumento el nombre o alias de este campo. Los valores han de ser numéricos, y devolverá el tipo de valores que se estén sumando. Si hay enteros con un *doublé,* devolverá un *doublé.* Si además hay un *long,* devolverá un *long.*

4.10. Cláusula GROUP BY

La cláusula GROUP BY define las expresiones de agrupación sobre las que se agregarán los resultados. Una expresión de agrupación utiliza un alias o una variable de identificación. La siguiente consulta cuenta el número de empleados en cada departamento:

Ejemplo 65. Cláusula GROUP BY

```
SELECT d.name, COUNT(e)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

Se debe tener en cuenta que el mismo campo utilizado en la cláusula SELECT se repite en la cláusula GROUP BY. Se puede aplicar más de una función agregada:

Ejemplo 66. Cláusula GROUP BY y funciones agregadas

```
SELECT d.name, COUNT(e), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

Esta variación de la consulta calcula el salario medio de todos los empleados de cada departamento, además de contar el número de empleados del departamento:

Ejemplo 67. Cláusula GROUP BY que agrupa por dos campos

```
SELECT d.name, e.salary, COUNT(p)
FROM Department d JOIN d.employees e JOIN e.projects p
GROUP BY d.name, e.salary
```

Ambas agrupaciones, por el nombre del departamento y por el salario del empleado, deben aparecer tanto en la cláusula SELECT como en la cláusula GROUP BY. Para cada departamento, esta consulta cuenta el número de proyectos asignados a los empleados en función de su salario.

En ausencia de una cláusula GROUP BY, las funciones agregadas se aplicarán a todo el resultado establecido como un solo grupo. Por ejemplo, la siguiente consulta devuelve el número de empleados y su salario medio en toda la empresa:

Ejemplo 68. Funciones agregadas en ausencia de GROUP BY

```
SELECT COUNT(e), AVG(e.salary)
FROM Employee e
```

4.11. Cláusula HAVING

La cláusula HAVING filtra los resultados de una consulta., funcionando de manera similar a la cláusula WHERE. La diferencia es que en la cláusula HAVING las expresiones condicionales se limitan en su mayoría a campos o agrupaciones de un solo valor.

Las condiciones de la cláusula HAVING también pueden utilizar funciones agregadas. En muchos aspectos, el uso principal de la cláusula HAVING es restringir los resultados basados en los valores totales de los resultados. La siguiente consulta utiliza esta técnica para recuperar todos los empleados asignados a dos o más proyectos:

Ejemplo 69. Cláusula HAVING

```
SELECT e, COUNT(p)
FROM Employee e JOIN e.projects p
GROUP BY e
HAVING COUNT(p) >= 2
```

4.12. Definición de consultas

JPA proporciona las interfaces `Query` y `TypedQuery` para configurar y ejecutar consultas. La interfaz `Query` se usa en aquellos casos en los que el tipo de resultado es Objeto o en consultas dinámicas cuanto el tipo de resultado puede no ser conocido. La interfaz `TypedQuery` es la preferida y puede usarse siempre que el tipo de resultado sea conocido. Como `TypedQuery` extiende a `Query`, una `TypedQuery` puede ser siempre tratada como una consulta simple, pero no viceversa.

Existen tres aproximaciones para definir una consulta en JP QL. Una consulta puede especificarse de manera dinámica en tiempo de ejecución, configurada en los metadatos de la unidad de persistencia (anotaciones o XML) y referenciadas por su nombre, o pueden ser especificadas de manera dinámica y guardadas para después ser referenciadas por su nombre. Las consultas dinámicas en JP QL no son más que cadenas, y por tanto pueden ser definidas sobre la marcha conforme se las necesita. Las consultas con nombre, por el contrario, son estáticas y no pueden ser cambiadas, pero más eficientes de ejecutar ya que el proveedor de persistencia puede traducir la cadena JP QL en SQL una vez que la aplicación se inicia en lugar de cada vez que se ejecuta la consulta. Definir dinámicamente una consulta y luego darle un nombre permite que una consulta dinámica se reutilice varias veces a lo largo de la vida de la aplicación, pero incurrir en el coste de procesamiento dinámico sólo una vez.

Las siguientes secciones comparan los enfoques y discuten cuándo se debe usar uno en lugar de los otros.

4.12.1. Definición de consultas dinámicas

Una consulta puede estar definida de forma dinámica pasando la cadena de consulta JP QL y el tipo de resultado esperado al método `createQuery()` de la interfaz *EntityManager*. El tipo de resultado puede omitirse para crear una consulta sin tipo definido. Esta aproximación se discutirá en secciones posteriores. No hay restricciones en la definición de la consulta. Todos los tipos de consulta JP QL están permitidos, así como el uso de parámetros. La capacidad de construir una cadena en tiempo de ejecución y utilizarlo para una definición de consulta es útil, especialmente para aplicaciones en las que el usuario puede especificar criterios complejos y la forma exacta de la consulta no puede conocerse de antemano.

Un factor a tener en cuenta con las cadenas de consultas dinámicas es el coste de traducción de la cadena JP QL a SQL para ejecución. Un motor de consultas típico tendrá que analizar la cadena JP QL en un árbol de sintaxis, obtener los metadatos del mapeo objeto-relacional para cada entidad en cada expresión, y después generar un equivalente en SQL. Para aplicaciones que involucran muchas consultas, el coste de rendimiento para el procesamiento dinámico de las consultas puede ser un problema.

Muchos motores de búsqueda almacenan en caché la traducción en SQL para usos posteriores, pero puede no funcionar si la aplicación no utiliza enlaces entre parámetros y concatena los valores de los

parámetros directamente en las cadenas de consulta. Como resultado, cada vez que una consulta requiere los parámetros, se construye una nueva.

El método *bean* mostrado en el Ejemplo 70 busca la información del salario teniendo como parámetros el nombre del departamento y el nombre del empleado. Existen dos problemas con este ejemplo, uno relacionado con el rendimiento y otro relacionado con la seguridad. Debido a que los nombres están concatenados en la cadena en lugar de utilizar los enlaces entre parámetros, cada vez se crea una nueva consulta. Si se hicieran cien llamadas al método se crearían cien cadenas de consulta distintas. Esto no sólo requiere un análisis excesivo de JP QL, sino que también dificulta que el proveedor de persistencia cree una caché de consultas ya convertidas.

Ejemplo 70. Definición dinámica de una consulta.

```
public class QueryService {
 @PersistenceContext(unitName="DynamicQueries")
 EntityManager em;
 public long queryEmpSalary(String deptName, String empName) {
        String query = "SELECT e.salary " +
                              "FROM Employee e " +
                       "WHERE e.department.name = '" + deptName +
                       "' AND " +
                       " e.name = '" + empName + "'";
        return em.createQuery(query, Long.class).getSingleResult();
 }
}
```

El segundo problema con este ejemplo es que es vulnerable frente a ataques de inyección de código, donde un usuario malicioso podría pasar un valor que altere la consulta a su gusto. La solución a este problema sería considerar un caso donde el argumento del departamento es fijado por la aplicación, pero el usuario sea capaz de especificar el nombre del empleado (el jefe del departamento está consultando los salarios de sus empleados, por ejemplo). Si el argumento del nombre fuese el texto `'_UNKNOWN' OR e.name = 'Roberts'`, la consulta real analizada por el motor de consulta sería la siguiente:

```
SELECT e.salary
FROM Employee e
WHERE e.department.name = 'NA65' AND
       e.name = '_UNKNOWN' OR
       e.name = 'Roberts'
```

Introduciendo la condición OR, el usuario se da a sí mismo acceso a los valores del salario de cualquier empleado en la empresa ya que la condición original `AND` tiene una precedencia mayor que `OR`, y es poco probable que el nombre falso del empleado pertenezca a un empleado real de ese departamento.

Este tipo de problemas pueden sonar improbables, pero en la práctica muchas aplicaciones web toman el texto enviado en una solicitud `GET` o `POST` y construyen ciegamente consultas de este tipo sin tener en cuenta los posibles efectos secundarios. Uno o dos intentos que resultan en un rastreo de pila de analizadores que se muestran en la página web y el atacante aprenderá todo lo que necesita saber sobre cómo alterar la consulta a su favor.

El Ejemplo 71 muestra el mismo método que el Ejemplo 70, pero en este caso usa parámetros con nombre. Esto no solo reduce el número de consultas generadas y analizadas por el motor de consulta, sino que también elimina la posibilidad de que la consulta sea alterada.

Ejemplo 71. Definición dinámica de una consulta utilizando parámetros.

```
public class QueryService {
 private static final String QUERY =
        "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND " +
        " e.name = :empName ";
 @PersistenceContext(unitName="QueriesUnit")
 EntityManager em;
 public long queryEmpSalary(String deptName, String empName) {
        return em.createQuery(QUERY, Long.class)
               .setParameter("deptName", deptName)
               .setParameter("empName", empName)
               .getSingleResult();
 }
}
```

El uso de enlaces entre parámetros mostrado en el Ejemplo 71 elimina el riesgo de seguridad descrito antes ya que la consulta original nunca es alterada. Los parámetros se organizan utilizando el JDBC API y son manejados directamente por la base de datos. El texto de una cadena de parámetro es citado por la base de datos, por lo que el ataque malicioso terminaría produciendo la siguiente consulta:

```
SELECT e.salary
FROM Employee e
WHERE e.department.name = 'NA65' AND
        e.name = '_UNKNOWN'' OR e.name = ''Roberts'
```

Las comillas simples utilizadas en el parámetro de consulta se han escapado al prefijarlas con una comilla simple adicional. Esto elimina cualquier significado especial y toda la secuencia es tratada como un único valor de cadena.

En general, es recomendable definir las consultas con nombre de manera estática, especialmente para consultas que son ejecutadas con frecuencia. Si son necesarias las consultas dinámicas, se ha de tener cuidado y utilizar los enlaces entre parámetros en lugar de concatenar valores de parámetros en las cadenas de consulta para minimizar el número de consultas analizadas por el motor de consulta.

4.12.2. Definición de Consultas con Nombre

Las consultas con nombre son una herramienta poderosa a la hora de organizar la definición de consultas y mejorar el rendimiento de la aplicación. Una consulta con nombre se define utilizando la anotación `@NamedQuery`, que puede ponerse en la definición de la clase para cualquier entidad. La anotación define el nombre de la consulta, así como el texto de la consulta. El Ejemplo 72 muestra cómo la cadena de consulta utilizada en el Ejemplo 71 se declararía como una consulta con nombre.

Ejemplo 72. Definición de una consulta con nombre.

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

Las consultas con nombre se colocan normalmente en la clase entidad que corresponde con el resultado de la consulta, así que la entidad Empleado sería una buena localización para esta consulta. Nótese el uso de concatenación de cadenas en la definición de la anotación. Dar formato a las consultas ayuda visualmente a la legibilidad de la definición de la consulta. La basura asociada normalmente a la repetición de cadenas concatenadas no se aplica en este caso ya que la anotación será procesada una única vez al comienzo y será ejecutada durante el tiempo de ejecución en forma de consulta.

El nombre de la consulta está englobado en la unidad de persistencia y debe ser único. Esta restricción es importante, ya que algunos nombres para las consultas se usan muy comúnmente, como es el caso de `"findAll"`, y cada uno deberá estar cualificado para cada entidad. Una práctica común es utilizar como prefijo del nombre de la consulta el nombre de la entidad. Por ejemplo, la consulta `"findAll"` para la entidad Empleado deberá llamarse `"Employee.findAll"`. No está muy claro que ocurriría si dos consultas englobadas en la misma unidad de persistencia tuvieran el mismo nombre, pero es probable que la implementación de la aplicación falle o que una consulta sobrescriba a la otra, lo que daría resultados impredecibles en el tiempo de ejecución.

Si se debe nombrar más de una consulta en una clase, todas deben englobarse mediante la anotación `@NamedQueries`, la cual acepta un *array* de una o varias anotaciones `@NamedQuery`. El Ejemplo 73 muestra la definición de varias consultas relacionadas con la entidad Empleado. Las consultan pueden también ser definidas (o redefinirse) utilizando XML.

Ejemplo 73. Definición de varias consultas con nombre en una entidad.

```
@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")
})
```

Ya que la cadena de consulta se define en la anotación, no puede ser modificada por la aplicación en el tiempo de ejecución. Esto contribuye al rendimiento de la aplicación y ayuda a prevenir los problemas de seguridad comentados en la sección anterior. Debido a la naturaleza estática de la cadena de consulta, cualquier criterio adicional requerido por la consulta debe ser especificado utilizando parámetros de consulta. El Ejemplo 74 muestra el uso de la llamada `createNamedQuery()` en la interfaz *EntityManager* para crear y ejecutar una consulta con nombre que requiere un parámetro de consulta.

Ejemplo 74. Ejecutando una consulta con nombre.

```
public class EmployeeService {
  @PersistenceContext(unitName="EmployeeService")
  EntityManager em;
  public Employee findEmployeeByName(String name) {
         return em.createNamedQuery("Employee.findByName",
                Employee.class)
                .setParameter("name", name)
                .getSingleResult();
         }
  // ...
}
```

Los parámetros con nombres son la opción más práctica para las consultas con nombre porque auto documentan de forma efectiva el código de la aplicación que llama a las consultas.

4.12.3. Definición dinámica de consultas con nombre

Una aproximación intermedia en la de crear las consultas de manera dinámica y después guardarlas como consultas con nombre en el `EntityManagerFactory`. En ese punto pasa a ser como cualquier otra consulta con nombre declarada de manera estática en los metadatos. Aunque pueda parecer una buena práctica, esto solo resulta útil en algunos casos específicos. La principal ventaja de esto se encuentra en el caso de que existan consultas que no se conocen hasta el tiempo de ejecución, y que son reeditadas

repetidamente. Una vez que la consulta dinámica se transforma en una consulta con nombre, sólo será procesada una vez. En la implementación se especifica si se debe procesar cuando la consulta se registra como consulta con nombre, o bien si se aplaza hasta la primera vez que se ejecuta.

Una consulta dinámica puede ser transformada en una con nombre utilizando el método `addNamedQuery()` del `EntityManagerFactory`. El Ejemplo 75 muestra cómo se hace.

Ejemplo 75. Creación de una consulta con nombre de manera dinámica.

```
public class QueryService {
 private static final String QUERY =
        "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND " +
        " e.name = :empName ";
 @PersistenceContext(unitName="QueriesUnit")
   EntityManager em;
 @PersistenceUnit(unitName="QueriesUnit")
 EntityManagerFactory emf;
 @PostConstruct
 public void init() {
        TypeQuery<Long> q = em.createQuery(QUERY, Long.class);
        emf.addNamedQuery("findSalaryForNameAndDepartment", q);
 }
 public long queryEmpSalary(String deptName, String empName) {
        return em.createNamedQuery("findSalaryForNameAndDepartment", Long.class)
                     .setParameter("deptName", deptName)
                     .setParameter("empName", empName)
                     .getSingleResult();
 }
 // ...
}
```

La inicialización del método *bean,* anotado con `@PostConstruct`, crea una consulta dinámica y le añade un conjunto de consultas con nombre. Como se ha mencionado anteriormente, las consultas con nombre están englobadas en la unidad de persistencia, así que tiene sentido que se añadan en el nivel del `EntityManagerFactory`.Requiere precaución la elección de los nombres, ya que añadir una consulta con el mismo nombre que una existente provocaría que se sobrescribiera la que ya existe. Debido a que se necesita el `EntityManagerFactory` en los dos métodos, se realiza una inyección en la instancia del *bean* utilizando `@PersisteneUnit`. También se podría haber accedido fácilmente desde el *EntityManager* utilizando el método `getEntityManagerFactory()`.

4.13. Tipos de Parámetros

Como se ha mencionado antes, JPA permite tanto parámetros con nombre como posicionales para las consultas en JP QL. Los valores de los parámetros son fijados en la interfaz `Query` utilizando el método `setParameter()`.

Existen tres variaciones de este método, tanto para los parámetros con nombre como para los parámetros posicionales. El primer argumento es siempre el nombre o número del parámetro. El segundo argumento es el objeto que se quiere enlazar con el parámetro. Los parámetros `Date` y `Calendar` además requieren un tercer argumento que especifique si el tipo pasado a JDBC es un valor `java.sql.Date`, `java.sql.Time` o `java.sql.TimeStamp`.

Considérese la siguiente definición de una consulta con nombre, que requiere dos parámetros con nombre:

```
@NamedQuery(name="findEmployeesAboveSal",
    query="SELECT e " +
            "FROM Employee e " +
            "WHERE e.department = :dept AND " +
            "   e.salary > :sal")
```

Esta consulta muestra uno de los buenos recursos de JP QL, que es que las entidades pueden ser utilizadas como parámetros. Cuando se traduce la consulta a SQL, las columnas asociadas con las *primary key* serán insertadas en la expresión condicional y apareadas con los valores de las *primary keys* de los parámetros. No es necesario saber cómo se mapea la *primary key* para escribir una consulta. El Ejemplo 76 muestra cómo enlazar los parámetros primitivos y los de la entidad requeridos en esta consulta.

Ejemplo 76. Enlazando parámetros con nombre.

```
public List<Employee> findEmployeesAboveSal(Department dept, long minSal) {
  return em.createNamedQuery("findEmployeesAboveSal", Employee.class)
          .setParameter("dept", dept)
          .setParameter("sal", minSal)
          .getResultList();
}
```

Un aspecto a tener en cuenta con los parámetros de consulta, es que el mismo parámetro puede ser utilizado varias veces en la cadena de consulta, pero solo necesita ser enlazado una vez utilizando el método `setParameter()`. Por ejemplo, considérese el siguiente ejemplo de la definición de una consulta con nombre, donde el parámetro `"dept"` se usa dos veces en la cláusula WHERE:

```
@NamedQuery(name="findHighestPaidByDepartment",
    query="SELECT e " +
            "FROM Employee e " +
            "WHERE e.department = :dept AND " +
            "  e.salary = (SELECT MAX(e.salary) " +
            "  FROM Employee e " +
            "  WHERE e.department = :dept)")
```

Para ejecutar esta consulta, el parámetro "`dept`" necesita ser fijado una única vez con `setParameter()`, como en el siguiente ejemplo:

```
public Employee findHighestPaidByDepartment(Department dept) {
 return em.createNamedQuery("findHighestPaidByDepartment",Employee.class)
        .setParameter("dept", dept)
        .getSingleResult();
}
```

4.14. Ejecutando Consultas

Las interfaces `Query` y `TypedQuery` proporcionan tres formas diferentes de ejecutar una consulta, dependiendo en si la consulta devuelve o no resultados y el de qué cantidad de resultados se esperan. Para consultas que devuelven valores, el desarrollador debe elegir entre llamar al método `getSingleResult()`, si la consulta espera un único resultado, o al método `getResultList()` si se espera más de un resultado. Ambas interfaces incorporan los mismos métodos, que difieren únicamente en el tipo de resultado que se devuelve.

La forma más simple de ejecución de consultas es mediante el método `getResultList()`. Éste devuelve una colección que contiene los resultados de la consulta. Se especifica el tipo de resultado como una `List` en lugar de una `Collection` para soportar consultas que especifiquen un orden de clasificación. Si la consulta utiliza la cláusula ORDER BY para especificar la ordenación, los resultados serán devueltos en una lista con el mismo orden. El 77 muestra cómo una consulta puede usarse para generar un menú para una aplicación de línea de comandos que muestra el nombre de cada empleado que trabaja en un proyecto, así como el nombre del departamento al que pertenece dicho empleado. Los resultados se ordenan por el nombre del empleado. Las consultas están desordenadas por defecto.

Ejemplo 77. Iteración sobre resultados ordenados.

```
public void displayProjectEmployees(String projectName) {
  List<Employee> result = em.createQuery(
        "SELECT e " +
        "FROM Project p JOIN p.employees e "+
        "WHERE p.name = ?1 " +
        "ORDER BY e.name",
```

```
            Employee.class)
                  .setParameter(1, projectName)
                  .getResultList();
      int count = 0;
      for (Employee e : result) {
            System.out.println(++count + ": " + e.getName() + ", " +
                  e.getDepartment().getName());
      }
}
```

El método `getSingleResult()` se proporciona para facilitar las consultas que devuelven un único valor. En lugar de iterar sobre el primer resultado de una colección, el objeto se devuelve directamente. Es importante darse cuenta de que `getSingleResult()` se comporta de manera distinta a `getResultList()` en cómo maneja resultados inesperados. Mientras que `getResultList()` devuelve una colección vacía cuando no hay resultados disponibles, `getSingleResult()` lanza una excepción `NoResultException`. Por tanto, si existe la posibilidad de que el resultado de la consulta no se encuentre, se debe manejar esta excepción.

Si varios resultados están disponibles tras ejecutar la consulta en lugar de un único valor esperado, `getSingleResult()` lanzará la excepción `NonUniqueResultException`. De nuevo, esto puede resultar un problema para el código de la aplicación si los criterios de consulta pueden dar lugar a que se devuelva más de un registro en determinadas circunstancias. Aunque es conveniente usar `getSingleResult()`, hay que estar seguro de que la consulta y sus posibles resultados se comprenden bien; si no, el código de la aplicación puede tener que lidiar con una excepción durante el tiempo de ejecución no esperada. A diferencia de otras excepciones lanzadas por las operaciones del *entity manager*, estas excepciones no harán que el proveedor deshaga la transacción actual, si la hay.

Los objetos `Query` y `TypedQuery` pueden reutilizarse cada vez que se necesiten siempre y cuando el contexto de persistencia donde se han creado las consultas siga activo. Para *entity manager* englobados en transacciones, el tiempo de vida de estos objetos está ligado al tiempo de vida de la transacción. Otros *entity manager* pueden reutilizarlos hasta que el *entity manager* que los creó se cierre o se elimine.

4.14.1. Trabajando con resultados de consulta

El tipo de resultado de una consulta es determinado por las expresiones listadas en la cláusula SELECT de la consulta. Si el tipo de resultado de una consulta es la entidad Empleado, entonces al ejecutar `getResultList()` se obtendrá una colección de cero o más instancias de la entidad Empleado. Existe una amplia variedad de resultados posibles, dependiendo de la composición de la consulta. Algunos de los tipos más frecuentes se listan a continuación:

- Tipos básicos, tales como *String*, primitivas de datos y objetos JDBC.
- Entidades
- Un *array* de `Object`
- Tipos definidos por el usuario mediante un constructor.

Para los desarrolladores familiarizados con JDBC, el punto más importante a tener en cuenta es que al usar las interfaces `Query` y `TypedQuery` los resultados no se encapsulan en un `ResultSet` de JDBC. La colección o el resultado único corresponden directamente con el tipo de resultado de la consulta.

Siempre que se devuelve una instancia de una entidad, este se gestiona mediante el contexto de persistencia activa. Si se modifica dicha instancia y el contexto de persistencia forma parte de la transacción, los cambios serán persistidos en la base de datos. La única excepción para esta regla es el uso de *entity managers* englobados en una transacción pero que actúan fuera de ella. Cualquier consulta que se ejecute en esta situación devuelve instancias de entidades independientes en lugar de las instancias de entidades gestionadas. Para realizar cambios sobre estas entidades independientes, deben incluirse en un contexto de persistencia antes de que puedan sincronizarse con la base de datos.

Una consecuencia de la gestión a largo plazo de entidades con contextos de persistencia extendidos y gestionados por la aplicación es que la ejecución de consultas de gran tamaño hará que el contexto de persistencia crezca, ya que almacena todas las instancias de entidades gestionadas que se devuelven. Si muchos de estos contextos de persistencia están ligados a un gran número de entidades gestionadas durante largos períodos de tiempo, el uso de la memoria puede convertirse en un problema. El método `clear()` de la interfaz *EntityManager* puede utilizarse para borrar contextos de persistencia extendida y administrada por la aplicación, eliminando las entidades administradas que sean innecesarias.

4.14.2. Transmisión de los resultados de una consulta

Como se comentó en el primer capítulo, uno de los cambios introducidos en JPA 2.2 es la capacidad de transmitir el resultado de la ejecución de una consulta. Para ello, el método `getResultStream()` de las interfaces `Query` y `TypedQuery` convierte los resultados en formato "*stream*". Usando el Ejemplo 77, se muestra el uso de este método para obtener los valores del salario en forma de *stream.*

```
public void displayProjectEmployees(String projectName) {
    final AtomicInteger salary = 0;
    List<Employee> result = em.createQuery(
         "SELECT e " +
         "FROM Project p JOIN p.employees e "+
         "WHERE p.name = ?1 " +
         "ORDER BY e.name",
         Employee.class)
                .setParameter(1, projectName)
     Stream<emp> empStream = result.getResultStream();
     empStream.forEach( e -> salary.set(salary.get() + e.getSalary()));
     return salary.get();
}
```

Este método resulta muy útil cuando se necesita procesar un conjunto muy grande de resultados.

4.14.3. Resultados sin tipo

Cuando el tipo de resultado de una consulta es `Object`, o la consulta JP QL selecciona múltiples objetos, se puede recurrir al uso de la versión sin tipo de los métodos de creación de consultas. Esto se hace omitiendo el tipo de resultado que genera una instancia de `Query` en lugar de una instancia de `TypedQuery`, lo que hará que el método `getResultList()` devuelva una `List` desvinculada y que el método `getSingleResult()` devuelva un `Object`.

4.14.4. Mejorando consultas de solo lectura

Cuando los resultados de una consulta no se van a modificar, consultas que usen *entity managers* englobados en una transacción pueden ser más útiles que aquellas consultas que se ejecuten dentro de la propia transacción si el tipo de resultado va a ser una entidad.

Cuando los resultados de una consulta estén preparados dentro de una transacción, el proveedor de persistencia toma su tiempo para convertir los resultados en entidades ya gestionadas. Esto suele conllevar un procesamiento de los datos para tener una base con la que comparar una vez que la transacción se haya efectuado. Si las entidades gestionadas nunca van a modificarse, no merece la pena convertir los resultados en entidades gestionadas.

Fuera de una transacción, el proveedor de persistencia es capaz de optimizar el proceso si se van a tomar directamente los resultados, pudiendo ahorrarse la creación de esas entidades tratadas. Sin embargo, esto no funciona con aplicaciones ya gestionadas o *entity managers* extendidos ya que su contexto de persistencia va más allá de la transacción. Cualquier resultado de alguna consulta que se encuentre en este tipo de contexto de persistencia debe ser modificada para una posterior sincronización con la base de datos, aunque no haya transacción.

Para encapsular operaciones de consulta, a través de un *bean* con transacciones de contenedor gestionado, la forma más fácil de llevar a cabo consultas sin transacción es usar el atributo de transacción NOT_SUPPORTED para el método de *bean* de sesión. Esto hará que cualquier transacción abierta se cierre, forzando a los resultados de la consulta de ser aislados y permitiendo esta mejora.

4.14.5. Tipos de Resultados Especiales

Siempre que una consulta involucre más de una expresión en la cláusula SELECT, el resultado de la consulta será una `List` de *arrays* `Object`. Ejemplos frecuentes incluyen la proyección de campos de entidad y consultas agregadas donde se usa la agrupación de expresiones o múltiples funciones.

El Ejemplo 78 revisa el generador de menú del Ejemplo 77 utilizando una consulta de proyección en lugar de devolver instancias completas de la entidad `Employee`. Cada elemento de la `List` se convierte en un *array* de `Object` que se usa después para extraer la información sobre el nombre del empleado y del departamento. Se utiliza una consulta sin tipo porque el resultado tiene varios elementos.

Ejemplo 78. Manejando múltiples tipos de resultados

```
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")
                .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]);
    }
}
```

4.14.6. Paginación de consultas

En ocasiones conjuntos grandes de resultados de las consultas son un problema para muchas aplicaciones. En casos en los que sería abrumador mostrar todo el conjunto de resultados, o si el medio de aplicación hace que la visualización de muchas filas sea ineficaz (aplicaciones web, en particular), las aplicaciones deben ser capaces de mostrar los rangos de un conjunto de resultados y proporcionar a los usuarios la capacidad de controlar el rango de datos que están visualizando. La forma más común de esta técnica es presentar al usuario una tabla de tamaño fijo que actúa como una ventana corrediza sobre el conjunto de resultados. Cada incremento de los resultados mostrados se llama página, y el proceso de navegación a través de los resultados se llama paginación.

La búsqueda eficiente de conjuntos de resultados ha sido durante mucho tiempo un desafío tanto para los desarrolladores de aplicaciones como para los proveedores de bases de datos. Antes de que existiera soporte a nivel de base de datos, una técnica común era recuperar primero todas las claves primarias para el conjunto de resultados y luego emitir consultas separadas para los resultados completos utilizando rangos de valores de clave primaria.

Más tarde, los proveedores de bases de datos añadieron el concepto de número lógico de fila a los resultados de la consulta, garantizando que, siempre y cuando el resultado fuera ordenado, se pudiera confiar en el número de fila para recuperar partes del conjunto de resultados. Más recientemente, la especificación JDBC ha llevado esto aún más lejos con el concepto de conjuntos de resultados desplazables, en los que se puede navegar hacia delante o hacia atrás según sea necesario.

Las interfaces `Query` y `TypedQuery` permiten la paginación mediante los métodos `setFirstResult()` y `setMaxResults()`. Estos métodos especifican el primer resultado que debe recibirse (numerado desde cero) y el número máximo de resultados a devolver en relación a ese punto. Los valores establecidos para estos métodos también se pueden recuperar mediante los métodos `getFirstResult()` y `getMaxResults()`. Usando los métodos `next()`, `previous()`, y `getCurrentResults()`, el código de presentación puede navegar por los resultados según sea necesario. El proveedor de persistencia puede elegir implementar el soporte para esta característica de varias maneras diferentes porque no todas las bases de datos se comportan de la misma manera. Es una buena idea familiarizarse con la forma en que el proveedor lleva a cabo la paginación y que nivel de soporte existe en la base de datos que se quiere utilizar para la aplicación.

4.14.7. Tiempo de Espera de las consultas

En general, cuando se ejecuta una consulta se bloquea hasta que la consulta de la base de datos retorna. Además de la preocupación obvia sobre las consultas desbordadas y la respuesta de la aplicación, también puede ser un problema si la consulta está participando en una transacción y se ha establecido un tiempo de espera en la transacción JTA o en la base de datos. El tiempo de espera en la transacción o base de datos puede hacer que la consulta se interrumpa antes de tiempo, pero también causará que la consulta realice un *rollback*, lo que impide que se siga trabajando en la misma transacción. Si una aplicación necesita establecer un límite en el tiempo de respuesta de la consulta sin causar un *rollback* de la transacción, la propiedad `javax.persistence.query.query.timeout` puede ser establecida en la consulta o también como parte de la unidad de persistencia. Esta propiedad define el número de milisegundos permitidos de ejecución de la consulta antes de que sea abortada.

El Ejemplo 79 muestra como fijar el valor del tiempo de espera para una consulta. Este ejemplo utiliza el mecanismo de sugerencias de consulta, que se comentará más adelante en el capítulo.

Ejemplo 79. Fijando el tiempo de espera de una consulta

```
public Date getLastUserActivity() {
    TypedQuery<Date> lastActive =
  em.createNamedQuery("findLastUserActivity", Date.class);
    lastActive.setHint("javax.persistence.query.timeout", 5000);
    try {
  return lastActive.getSingleResult();
    } catch (QueryTimeoutException e) {
  return null;
    }
}
```

Desafortunadamente, fijar el tiempo de espera de las consultas no es un comportamiento portable. Puede que no esté soportado por todas las plataformas de bases de datos y tampoco por todos los proveedores de persistencia. Por tanto, las aplicaciones que decidan utilizar tiempos de respuesta en las consultas deben estar preparadas para tres posibles escenarios. El primero es que puede que la propiedad sea ignorada y no produzca ningún efecto. El segundo es que la propiedad se habilite, de manera que cualquier operación SELECT, DELETE o UPDATE que se realice fuera del tiempo de espera será abortada y se lanzará una excepción `QueryTimeoutException`, pero no producirá un *rollback* sobre las operaciones. El Ejemplo 79 demuestra una manera de manejar esta excepción. El tercer escenario posible es que la propiedad se habilite, pero esto provoca que la base de datos fuerce un *rollback* en la transacción cuando se excede el tiempo de espera. En este caso, se lanzará una `PersistenceException` y la transacción será marcada para *rollback*. En general, si la propiedad está activada, la aplicación debe ser escrita para manejar `QueryTimeoutException`, pero no debe fallar si se excede el tiempo de espera y no se lanza la excepción.

4.15. UPDATE y DELETE masivo

Como sus homólogos en SQL, las sentencias JP QL UPDATE y DELETE masivas están diseñadas para realizar cambios a un gran número de entidades en una única operación sin requerir que las entidades individuales sean recuperadas y modificadas utilizando el *EntityManager*. De manera distinta a SQL, que opera con tablas, las sentencias UPDATE y DELETE de JP QL deben tener en cuenta el rango completo del mapeo de la entidad. Estas operaciones suponen un reto para los proveedores a la hora de implementarlas correctamente, lo que da lugar a una serie de restricciones en su uso que deben ser comprendidas por los desarrolladores. Las siguientes secciones describen cómo utilizar estas operaciones de forma eficaz y los problemas que pueden surgir cuando se utilizan de forma incorrecta.

4.15.1. Consultas de actualización (UPDATE)

Las consultas de actualización proporcionan un equivalente a la sentencia UPDATE de SQL, pero con expresiones condicionales JP QL. La forma de una consulta de actualización es la siguiente:

```
UPDATE <entity name> [[AS] <identification variable>]
SET <update_statement> {, <update_statement>}*
[WHERE <conditional_expression>]
```

Cada instrucción UPDATE consta de un alias o nombre de campo de un solo valor, la asignación de los valores correspondientes a la expresión anterior por medio de un SET y una condición.

La asignación tiene ciertas restricciones en comparación con las condiciones típicas. El lado derecho de la asignación debe resolverse con una expresión literal y simple, que resuelva un tipo básico, una función, una variable de identificación o un parámetro de entrada. El tipo de resultado de esa expresión debe ser compatible con el campo del lado izquierdo de la asignación.

El siguiente ejemplo muestra cómo realizar una consulta de actualización al cambiar el salario de los empleados que ganan 55000 dólares a 60000.

```
UPDATE Employee e
SET e.salary = 60000
WHERE e.salary = 55000
```

La cláusula WHERE de una instrucción UPDATE funciona igual que una instrucción SELECT y puede utilizar el campo definido en la cláusula UPDATE. Una actualización un poco más compleja sería otorgar un aumento de $5,000 a los empleados que trabajaron en un proyecto en particular:

```
UPDATE Employee e
SET e.salary = e.salary + 5000
WHERE EXISTS (SELECT p
              FROM e.projects p
```

```
                WHERE p.name = 'Release2')
```

Se puede modificar más de una propiedad de la entidad de destino con una única instrucción UPDATE. Por ejemplo, la siguiente consulta actualiza la central telefónica para empleados en la ciudad de Ottawa y cambia la terminología del tipo de teléfono:

```
UPDATE Phone p
SET p.number = CONCAT('288', SUBSTRING(p.number, LOCATE('-', p.number), 4)),
      p.type = 'Business'
WHERE p.employee.address.city = 'Ottawa' AND p.type = 'Office'
```

4.15.2. Consultas de borrado (DELETE)

La consulta de borrado proporciona la misma capacidad que la instrucción DELETE de SQL, pero con condiciones JP QL. La forma de borrar una consulta es la siguiente:

```
DELETE FROM <entity name> [[AS] <identification variable>]
[WHERE <condition>]
```

El siguiente ejemplo elimina todos los empleados que no están asignados a un departamento:

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

La cláusula WHERE para una instrucción DELETE funciona igual que para una instrucción SELECT. Todas las condiciones están disponibles para filtrar el conjunto de valores a eliminar. Si no se proporciona la cláusula WHERE, se eliminan todos los valores de esa entidad.

También se eliminarán las instancias de las subclases de esas entidades que cumplan los criterios de esa condición. Sin embargo, las consultas de borrado no respetan las reglas en cascada. Ninguna otra entidad que no sea el tipo al que se hace referencia en la consulta y sus subclases podrá ser eliminada, incluso si la entidad tiene relaciones con otras entidades con la función de eliminación en cascada habilitado.

4.15.3. Uso de UPDATE y DELETE masivo

Una vez entendido cómo realizar consultas de tipo UPDATE y DELETE, es necesario conocer cómo se ejecutan. El procedimiento es muy similar al resto de consultas vistas hasta ahora, pero, en este caso, no se recupera ningún dato o lista de datos. Para ejecutar las consultas de este tipo se utiliza el método `executeUpdate()`, como puede verse en los Ejemplos 80 y 81.

Ejemplo 80. Modificación masiva de entidades.

```
public class EmployeeService {
    @PersistenceContext(unitName="BulkQueries")
    EntityManager em;
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    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();
    }
}
```

Ejemplo 81. Borrado masivo de entidades.

```
public class ProjectService {
  @PersistenceContext(unitName="BulkQueries")
  EntityManager em;
  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public void removeEmptyProjects() {
        em.createQuery("DELETE FROM Project p " +
                            "WHERE p.employees IS EMPTY")
                .executeUpdate();
  }
}
```

El primer problema a tener en cuenta cuando se utilizan estas sentencias es que el contexto de persistencia no se actualiza para reflejar los resultados de la operación. Las operaciones masivas se emiten como SQL en la base de datos, evitando las estructuras en memoria del contexto de persistencia. Por tanto, modificar el salario de todos los empleados no cambiará los valores actuales de ninguna entidad que se maneje en memoria como parte del contexto de persistencia. El desarrollador sólo puede confiar en las entidades recuperadas después de que la operación masiva se haya completado.

Cuando se utilizan contextos de persistencia englobados en la transacción, la operación masiva debe ejecutarse en una transacción por sí misma o ser la primera operación en la transacción. Ejecutar la operación masiva en su propia transacción es preferible porque minimiza la posibilidad de obtener datos accidentalmente antes de que ocurra el cambio masivo. Ejecutar la operación masiva y después trabajar con entidades tras haberse completado es también una opción segura, ya que cualquier operación `find()` o consulta accederá a la base de datos y obtendrá los resultados actualizados. Los Ejemplos 80 y 81

utilizan el atributo de transacción `REQUIRES_NEW` para asegurar que las operaciones masivas ocurren en sus propias transacciones.

Una estrategia típica para los proveedores de persistencia a la hora de tratar con operaciones masivas es invalidar cualquier caché de memoria de los datos relacionados con la entidad objetivo. Esto obliga a que los datos se obtengan de la base de datos la próxima vez que se necesiten. La cantidad de datos en caché que se invalidan depende de la sofisticación del proveedor de persistencia. Si el proveedor puede detectar que la modificación solo afecta un rango pequeño de entidades, esas entidades específicas pueden ser invalidadas, dejando otros datos en caché inalterados. Estas optimizaciones son limitadas y si el proveedor no puede estar seguro del alcance del cambio, todo el caché será invalidado. Esto puede tener gran impacto en el rendimiento de la aplicación si se realizan con frecuencia cambios masivos.

PRECAUCIÓN: Las operaciones SQL UPDATE y DELETE no deben ejecutarse en tablas mapeadas por una entidad. Las operaciones JP QL le indican al proveedor qué estado de la entidad en caché debe ser invalidado para mantener la coherencia con la base de datos.

Las operaciones SQL nativas evitan estas comprobaciones y pueden llevar rápidamente a situaciones en las que la caché en memoria no está actualizada con respecto a la base de datos.

El peligro que conllevan las operaciones masivas, y la razón por la que deben ocurrir en primer lugar en una transacción, es que cualquier entidad gestionada activamente por un contexto de persistencia permanecerá así, ajena a los cambios reales que se produzcan a nivel de la base de datos. El contexto de persistencia activo es independiente y distinto de cualquier caché de datos que el proveedor pueda utilizar para las optimizaciones.

4.16. Sugerencias de consulta (Query Hints)

Las sugerencias (conocidas como `hints` en inglés) de consulta son el punto de extensión de JPA para las funciones de consulta. Una sugerencia es simplemente un nombre de cadena y un valor de objeto. Las sugerencias permiten añadir recursos a JPA sin introducir un nuevo API. Esto incluye recursos estándar tales como los tiempos de espera de consulta, así como características específicas del proveedor. Cuando las sugerencias no están incluidas en la especificación de JPA, aunque tengan el mismo nombre, no serán portables entre proveedores. Cada consulta puede asociarse con cualquier número de sugerencias, especificadas tanto en los metadatos de la unidad de persistencia como parte de la anotación `@NamedQuery`, como en las interfaces `Query` o `TypedQuery` utilizando el método `setHint()`. Las sugerencias que se han asociado a una consulta pueden recuperarse utilizando el método `getHints()`, que devuelve un mapa de pares nombre-valor.

Para simplificar la portabilidad entre proveedores, los proveedores de persistencia están programados para ignorar aquellas sugerencias que no entienden. El Ejemplo 82 muestra la sugerencia "`eclipselink.cache-usage`" soportado por la *JPA Reference Implementation* para indicar que el caché no debe comprobarse cuando se lea un Empleado de la base de datos. A diferencia del método `refresh()` de la interfaz *EntityManager*, esta sugerencia no hará que el resultado de la consulta sobrescriba el valor de la caché actual.

Ejemplo 82. Utilizando sugerencias de consulta

```
public Employee findEmployeeNoCache(int empId) {
 TypedQuery<Employee> q = em.createQuery(
        "SELECT e FROM Employee e WHERE e.id = :empId", Employee.class);
 // force read from database
 q.setHint("eclipselink.cache-usage", "DoNotCheckCache");
 q.setParameter("empId", empId);
 try {
        return q.getSingleResult();
 } catch (NoResultException e) {
        return null;
 }
}
```

Si esta consulta fuese ejecutada con frecuencia, una consulta con nombre sería más eficiente. La siguiente definición de consulta con nombre incorpora la caché de las recomendaciones utilizada en el ejemplo:

```
@NamedQuery(name="findEmployeeNoCache",
    query="SELECT e FROM Employee e WHERE e.id = :empId",
            hints={@QueryHint(name="eclipselink.cache-usage",
                              value="DoNotCheckCache")})
```

El elemento `hint` acepta un *array* de anotaciones `@QueryHint`, permitiendo que se establezca cualquier número de sugerencias para una consulta. No obstante, una limitación del uso de anotaciones para la consulta nombrada es que las sugerencias están restringidas a valores que son cadenas, mientras que cuando se usa el método `Query.setHint()` se puede pasar cualquier tipo de objeto como valor de sugerencia. Esto puede ser particularmente relevante cuando se utilizan sugerencias propias de un proveedor.

4.17. Buenas Prácticas

Una aplicación típica con JPA tendrá definidas muchas consultas. En las aplicaciones empresariales, lo normal es consultar constantemente la información de la base de datos para cualquier uso, desde informes hasta listas desplegables en la interfaz del usuario. Así pues, el uso eficiente de las consultas puede tener un impacto importante en el rendimiento y la capacidad de respuesta de la aplicación. En esta sección se incluyen algunas buenas prácticas a tener en cuenta en el uso de consultas en cuanto a rendimiento se refiere.

4.17.1. Consultas con Nombre

Primero y, ante todo, se recomienda el uso de consultas con nombre siempre que sea posible. Los proveedores de persistencia suelen tomar medidas para pre compilar consultas JP QL con nombre a SQL como parte de la fase de implementación o inicialización de una aplicación. Esto evita la sobrecarga de analizar continuamente JP QL y generar sentencias SQL. Incluso con una caché para consultas ya convertidas, la definición de consultas dinámicas será menos eficaz que el uso de consultas con nombre.

Las consultas con nombre también proporcionan la mejor manera de utilizar los parámetros de consulta. Los parámetros de consulta ayudan a mantener al mínimo las distintas consultas SQL que son analizadas por la base de datos. Ya que las bases de datos suelen tener a mano una caché de las sentencias SQL para consultas que se usan con frecuencia, esta es una parte esencial para garantizar un rendimiento máximo de la base de datos.

Además, como se ha comentado en secciones anteriores, los parámetros de consulta ayudan a evitar fallos en la seguridad provocados por la concatenación de valores en las cadenas de consulta. Para aplicaciones Web, la seguridad tiene que ser una preocupación en todos los niveles de una aplicación. Se puede gastar mucho esfuerzo en intentar validar los parámetros de entrada o bien se pueden utilizar los parámetros de consulta y dejar que la base de datos haga el trabajo.

Cuando se nombran consultas, se debe decidir una estrategia para poner los nombres antes del desarrollo de la aplicación, teniendo en cuenta que el espacio de nombres de la consulta es global para cada unidad de persistencia. Si no se establece un patrón de nomenclatura, pueden surgir problemas con los nombres de las distintas consultas, por lo que es conveniente y recomendado añadir como prefijo de la consulta el nombre de la entidad a la que está dirigida, separado con un punto.

La habilidad de crear consultas con nombre de manera dinámica en código puede ser útil si ninguno de los casos anteriores se puede aplicar, o si se da algún caso en el que resulta relevante la creación de consultas dinámicas, pero en general es preferible y más seguro declarar todas las consultas con nombre de manera estática.

4.17.2. Sugerencias propias de los proveedores

Es común que los proveedores ofrezcan al desarrollador una gran variedad de sugerencias para permitir diferentes optimizaciones de rendimiento en las consultas. Las sugerencias de consultas pueden ser una herramienta esencial para cumplir con las expectativas de rendimiento. Si la portabilidad del código fuente a varios proveedores es importante, se debe evitar implementar estas sugerencias en el código de la aplicación. La localización ideal para las sugerencias de consulta es un archivo de mapeo XML o, como último recurso, como parte de la definición de una consulta con nombre. Las sugerencias suelen ser muy dependientes de la plataforma que se utiliza y es posible que haya que modificarlos con el tiempo, ya que los diferentes aspectos de la aplicación afectan al equilibrio general del rendimiento. Evitar la implementación de sugerencias propias del proveedor en el código si es posible.

4.17.3. UPDATE y DELETE masivas

Si se deben realizar operaciones de UPDATE y DELETE masivos, se debe asegurar que se realicen únicamente en una transacción aislada donde no se realicen más cambios. Hay muchas formas en las que estas consultas pueden afectar negativamente un contexto de persistencia activo. Intercalar estas consultas con otras operaciones que no impliquen tratamiento de datos masivos requiere una gestión cuidadosa por parte de la aplicación.

El versionado y bloqueo de entidades requiere una consideración especial cuando se realizan operaciones UPDATE masivas. Las operaciones DELETE masivas pueden tener un amplio rango de ramificaciones dependiendo de lo bien que el proveedor de persistencia pueda reaccionar y ajustar el almacenamiento de entidades en respuesta. Así pues, las operaciones de borrado y modificado masivas deben ser utilizadas con cuidado.

4.17.4. Diferencias entre proveedores

Aunque entender SQL no es necesario para escribir consultas en JP QL, conocer lo que sucede en respuesta a varias operaciones JP QL es una parte esencial del ajuste de rendimiento. Los `"joins"` en JP QL no siempre son explícitos, y puede sorprender lo complejo del SQL generado por una, aparentemente simple, consulta de JP QL.

Los beneficios de los recursos como la paginación de consultas dependen mucho también de la aproximación utilizada por el proveedor de persistencia. Hay una variedad de diferentes técnicas para conseguir la "paginación", muchos de los cuales sufren problemas de rendimiento y escalabilidad. Debido a que JPA no puede establecer una aproximación particular que funcione correctamente en todos los casos, es importante familiarizarse con la aproximación utilizada por el proveedor y si es o no configurable.

Finalmente, entender la estrategia del proveedor para cuándo y con qué frecuencia descarga el contexto de persistencia es necesario antes de considerar las optimizaciones, tales como cambiar el modo de descarga. Dependiendo de la arquitectura de almacenamiento y las optimizaciones de consulta utilizadas por el proveedor, cambiar el modo de descarga puede o no hacer una diferencia en la aplicación.