***CONSEJERÍA DE SANIDAD***

***PLAN DE FORMACIÓN 2013***

ÍNDICE

[Presentación del curso 4](#_Toc286263156)

[Requisitos del curso 4](#_Toc286263157)

[Introducción 5](#_Toc286263158)

[Historia de la persistencia en Java 5](#_Toc286263159)

[Algunas implementaciones de JPA 6](#_Toc286263160)

[Definiciones 7](#_Toc286263161)

[Definiciones básicas 7](#_Toc286263162)

[ManagedEntity 7](#_Toc286263163)

[EntityManagerFactory 7](#_Toc286263164)

[EntityManager 7](#_Toc286263165)

[PersistenceUnit 7](#_Toc286263166)

[Entidades 8](#_Toc286263167)

[Definición 8](#_Toc286263168)

[Metadatos de entidad 8](#_Toc286263169)

[Anotaciones Jpa 9](#_Toc286263170)

[@Entity 9](#_Toc286263171)

[@Table 9](#_Toc286263172)

[@Id 10](#_Toc286263173)

[@EmbeddedId 10](#_Toc286263174)

[@GeneratedValue 11](#_Toc286263175)

[@SecuenceGenerator 11](#_Toc286263176)

[@Column 12](#_Toc286263177)

[@JoinColumn 12](#_Toc286263178)

[@JoinTable 13](#_Toc286263179)

[@ManyToOne 13](#_Toc286263180)

[@OneToOne 13](#_Toc286263181)

[@OneToMany 14](#_Toc286263182)

[@ManyToMany 14](#_Toc286263183)

[@Temporal 15](#_Toc286263184)

[@NamedQuery 15](#_Toc286263185)

[@NamedQuerys 15](#_Toc286263186)

[Configuración de la persistencia 17](#_Toc286263187)

[Ficheros de configuración de persistencia 17](#_Toc286263188)

[El fichero persistence.xml 17](#_Toc286263189)

[JPQL 19](#_Toc286263190)

[JPQL BASICO 19](#_Toc286263191)

[SENTENCIAS CONDICIONALES 20](#_Toc286263192)

[PARAMETROS DINAMICOS 20](#_Toc286263193)

[ORDENAR LOS RESULTADOS 21](#_Toc286263194)

[OPERACIONES DE ACTUALIZACION 21](#_Toc286263195)

[OPERACIONES DE BORRADO 21](#_Toc286263196)

[CONSULTAS CON NOMBRE (ESTATICAS) 21](#_Toc286263197)

[CONSULTAS NATIVAS SQL 22](#_Toc286263198)

[Prácticas 24](#_Toc286263199)

[Configuración común a todas las prácticas 24](#_Toc286263200)

[Práctica 1 25](#_Toc286263201)

[Modelado de un chat 25](#_Toc286263202)

[Generación de las clases 27](#_Toc286263203)

[Persistencia de objetos 30](#_Toc286263204)

[Obtención del objeto EntityManagerFactory 30](#_Toc286263205)

[Creación de un objeto EntityManager. 31](#_Toc286263206)

[Operaciones de persistencia (inserción, borrado, consulta). 31](#_Toc286263207)

[Liberación del objeto EntityManager. 32](#_Toc286263208)

[Liberación del objeto EntityManagerFactory 32](#_Toc286263209)

[Modelado de una empresa 34](#_Toc286263210)

Presentación del curso

En los últimos tiempos se han realizado avances significativos en la simplificación de los desarrollos Java de aplicaciones empresariales. La Junta de Castilla y León sigue anclada en tecnologías de desarrollo muy desfasadas con respecto a las actuales tecnologías. Este curso pretende ayudar a los desarrolladores de aplicaciones en el uso de nuevas tecnologías que simplifican mucho los desarrollos software.

Las aplicaciones que desarrollamos habitualmente se componen de tres grandes bloques diferenciados: La capa de vista, la lógica de negocio y la capa acceso a datos. Este curso se orienta a la capa de vista y a la de acceso a datos. *JSF* da solución a la capa de vista mientras que JPA lo hace en la capa de acceso a datos.

El enfoque de este curso es eminentemente práctico. El curso se divide en dos grandes bloques, uno referido a JPA y el otro a *JSF*.

En el primer bloque ap*render*emos como modelar entidades persistentes en base de datos, como crearla, como acceder a ellas, como modificarlas y como eliminarlas.

En el segundo ap*render*emos a crear interfaces de usuario basadas en estándar *JSF*, a utilizar platillas reutilizables, a utilizar controles mejorados y a darlas soporte Ajax.

# Requisitos del curso

Este curso está dirigido a desarrolladores de aplicaciones J2EE y tiene un enfoque práctico. Por ello se asume que los alumnos tienen conocimientos de programación Java y JSP.

También se presupone que los alumnos tienen conocimientos de modelado de datos relacionales y lenguaje SQL.

El cambio tecnológico nos impone el uso de la versión 5 de Java. El uso de esta versión no supone apenas cambios en el desarrollo aunque utilizaremos la capacidad de anotaciones de esta versión en la parte de JPA. Al final del libro hay un apéndice con un resumen de las características de Java 5 que vamos a utilizar y ejemplos de cada una de ellas.

Es impensable en la actualidad el trabajar sin la ayuda de algún IDE de desarrollo que nos facilite la labor. Históricamente en la Junta de Castilla y León se ha utilizado JDeveloper en cualquiera de sus versiones. En este curso no utilizaremos dicho IDE sino que utilizaremos Eclipse. Este IDE además de soportar plenamente las tecnologías que vamos a ver en este curso, siendo además mucho más versátil.

Introducción

# Historia de la persistencia en Java

Desde los comienzos de la programación en Java han aparecido tecnologías que permitan el acceso y manipulación (persistencia) de datos ubicados en bases de datos relacionales. Un breve repaso de las tecnologías de persistencia sería el siguiente:

JDBC

La especificación JDBC (Java Database Conectivity) permitió la estandarización del acceso a las bases de datos. Siempre que hubiese un driver compatible JDBC para una base de datos, la especificación nos permite el acceso a la misma desde aplicaciones Java. El problema es que aunque JDBC es un estándar, SQL no lo es. El código SQL cambia para bases de datos distintas (p.e. MySQL y Oracle tienen sintaxis JAVA ligeramente distintas).

EJBs

Se introdujo en la primera versión de J2EE (Java 2 Enterprise Edition) como nueva solución al problema de la persistencia en forma de Entity Bean. Delegaban la persistencia al contenedor aunque con carencias en cuanto a portabilidad (configuración XML ad hoc en los despliegues para proveedores específicos), coste de red elevado por el acceso RMI a los beans, mapeo insuficiente de las relaciones entre Entity Beans (foreign keys), etc…

JDO

Se trató de un esfuerzo independiente por dar solución al problema de la persistencia. Inicialmente requería bases de datos orientadas a objetos aunque posteriormente se amplió a las bases de datos relacionales. Requiere de un proceso de “enhancement” del byte code generado por el compilador java que añade datos para la gestión de la persistencia en un proceso posterior a la compilación. También define un lenguaje de consulta orientado a objetos. JDO alcanzó el status de extensión JDK aunque nunca el status de estándar Java.

JPA

JPA es el estándar de persistencia desarrollado para la plataforma J2EE mediante el estándar EJB3. A diferencia de JDO, JPA es un estándar Java (JPA 1.0 [JSR220](http://jcp.org/en/jsr/detail?id=220) y JPA 2.0 [JSR317](http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html)) existiendo una serie de implantaciones tanto comerciales como libres de dicho estándar.

JPA permite mapear los objetos y relaciones entre los mismos a tablas relacionales, permitiendo utilizar POJOs (Plain Old Java Objects) para mantener las ventajas de la orientación a objetos en el acceso a la base de datos.

# Algunas implementaciones de JPA

Algunas de las implementaciones de JPA más usuales son:

[Hibernate](http://es.wikipedia.org/wiki/Hibernate)

[TopLink](http://www.oracle.com/technology/products/ias/toplink/index.html)

[CocoBase](http://www.thoughtinc.com)

[EclipseLink](http://www.eclipse.org/eclipselink/)

[OpenJPA](http://incubator.apache.org/openjpa/)

[Kodo](http://www.bea.com/kodo/)

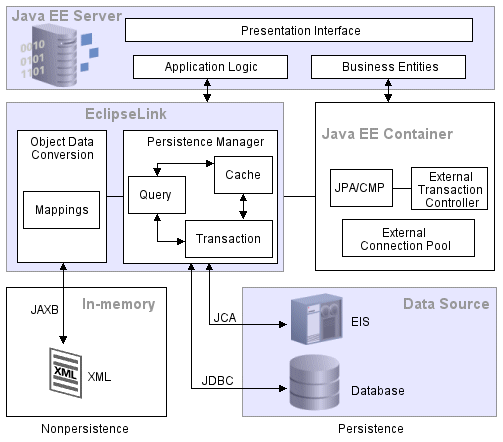
[JPOX](http://www.jpox.org/)

[Amber](http://www.caucho.com/resin-3.1/doc/amber.xtp)

EclipseLink JPA

En este seminario utilizaremos esta implementación para las prácticas de desarrollo. La versión que utilizaremos es la versión 2.0.1 correspondiente a la versión 2.0 de la especificación JPA. Los paquetes necesarios para la compilación y despliegues de aplicaciones se pueden obtener de [aquí](http://www.eclipse.org/eclipselink/downloads/index.php).

Figura 1: Modelo de arquitectura runtime de EclipseLink



Definiciones

# Definiciones básicas

## ManagedEntity

Entidad gestionada por un EntityManager. Se trata de todas las clases que se han declarado como entidades.

## EntityManagerFactory

Objeto que se utiliza para interactuar con la base de datos definida en una unidad de persistencia. Normalmente se puede identificar con un DataSource. Es la encargada de crear los EntityManagers.

## EntityManager

Gestor de entidades persistentes. Recupera, actualiza y mantiene sincronizados los ManagedEntities con la base de datos.

## PersistenceUnit

Elemento de agrupación lógica de entidades persistentes que deben de ser tratadas de forma diferente. Incluye:

* Un EntityManagerFactory y todos sus EntityManagers junto con su información de configuración.
* El conjunto de clases gestionadas incluidas en la unidad de persistencia gestionadas por los EntityManagers del EntityManagerFactory..
* Metadatos de mapeo que especifica como se mapean las clases en la base de datos.

Entidades

# Definición

Las entidades son objetos que tienen atributos y relaciones con otras entidades. En Java, los objetos son entidades.

Persistencia

La característica más importante de las entidades es que tienen persistencia. Esto significa que el estado, definido como el valor del conjunto de sus atributos puede ser recuperado en un instante posterior, más allá incluso del fin del proceso que creó la entidad.

Identidad

Otra característica de las entidades es que tienen una identidad (cualquier objeto java la tiene). Es más, posee una identidad persistente definida por el valor de un conjunto de los atributos que identifican unívocamente a la entidad.

Transaccionalidad

Las entidades son transaccionales. Solo pueden ser creadas, modificadas y borradas dentro de una transacción, y se requiere una transacción para que los cambios en la entidad puedan ser grabados en la base de datos. Los objetos java normales no cumplen esta restricción ya que pueden ser libremente modificados y solo obedecen las reglas que impone la máquina virtual

# Metadatos de entidad

Se asocia a cada entidad una serie de metadatos para describirla. Esta descripción debe de ser suficiente para permitir la persistencia de la entidad. La cantidad de metadatos necesarios para describir una entidad es mínima como ya veremos. Aunque se puede extender ese conjunto mínimo para detallar hasta el último detalle de la persistencia de la entidad. Existen dos modos definir los metadatos– anotaciones o XML.

Anotaciones

Este va a ser el modo que veremos para describir los metadatos de una entidad. Se basa en la capacidad de anotaciones que nos permiten las versiones de Java 5 y superiores.

Las anotaciones más frecuentes que utilizaremos son las de *@Entity, @Table, @Id, @Column, @JoinColumn, @JoinTable , @ManyToOne, @OneToOne, @OneToMany, @ManyToMany, @Temporal,@NamedQuery y @NamedQuerys*.

Anotaciones

Las anotaciones son un mecanismo para dotar a las clases de Meta información o auto información. Estas ya existían en versiones de java anteriores y de hecho se usaban, por ejemplo para generar el javaDoc (@Deprecated). Pero desde la versión 5 se han convertido en una parte Tipada del lenguaje y podemos trabajar con ellas tanto en tiempo de compilación como en tiempo de ejecución.

Anotar una porción de código nos sirve para procesarla de alguna manera en algún script, herramienta de despliegue o de diseño, para hacer uso de ellas en un framework, para trabajarlas distintas por un compilador…

Las anotaciones se pueden trabajar en tiempo de ejecución. La forma de hacerlo sería usando el API de reflection, que se ha completado en Java 5 para tener acceso a las anotaciones.

Un ejemplo de anotación sería

…

@Column(name=”C\_EMPRESA\_ID”, unique=true, length=10)

private Integer cEmpresaId;

…

La anotación (en este caso @Column) se aplica a la declaración del atributo cEmpresaId. Las anotaciones no significan nada salvo para los gestores que por reflexión acceden a ellas y obtienen información de las mismas.

XML

En este curso no veremos esta forma de declarar los metadatos. Lo único que diremos es que todo lo que se puede declarar mediante anotaciones se puede hacer utilizando un fichero XML. Esto nos proporciona compatibilidad con las versiones de Java inferiores a la versión 5.

# Anotaciones Jpa

Veamos la definición de las anotaciones JPA más habituales:

## @Entity

La anotación Entity se utiliza para declarar una clase Java como entidad. Precede a la definición de la clase y no admite atributos.

Ejemplo:

@Entity

public class Empresa {

…

## @Table

La anotación @Table se utiliza para declarar la tabla a la que se mapea una entidad. Precede a la definición de la clase y los atributos más utilizados son:

**schema** Nombre del esquema de BBDD de la tabla

**name** Nombre de la tabla

Ejemplo

@Entity

@Table(schema=”SCH01”, name=”T\_EMPRESA\_ID”)

public class Empresa {

…

## @Id

La anotación @Id se utiliza para declarar un atributo como identificador persistente de una entidad (primary key). Precede a la declaración de un atributo de la clase y no admite atributos.

Ejemplo

@Id

prívate Integer id;

## @EmbeddedId

La anotación @EmbeddedId se utiliza para declarar que la entidad posee un indicador de entidad múltiple (primary-key de varios campos). Precede a la declaración de un atributo de la clase que debe cuyo tipo debe de reunir una serie de requisitos y no admite atributos.

Ejemplo:

@EmbeddedId

prívate DepartamentoPK id;

Los requisitos que debe reunir el tipo del atributo al que anteponemos la anotación @EmbeddedId son:

* La clase debe de estar anotada como @Embeddable
* Debe de ser serializable
* Los atributos deben estar anotados con @Column para referenciar las columnas de la tabla que constituyen la clave.
* Debe de tener un constructor por defecto (sin parámetros).
* Debe de implementar los métodos equals y hashCode para la comparación de claves.

Un ejemplo de clase que valdría como embedded id sería:

@Embeddable

**public** **class** DepartamentoPK **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = 1L;

@Column(name = "C\_EMPRESA\_ID", unique = **true**, nullable = **false**)

**private** Long cEmpresaId;

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

@Column(name = "C\_DEPARTAMENTO\_ID", unique = **true**, nullable = **false**)

**private** Long cDepartamentoId;

**public** DepartamentoPK() {

}

**public** DepartamentoPK(Long empresaId, Long departamentoId) {

cEmpresaId = empresaId;

cDepartamentoId = departamentoId;

}

**public** DepartamentoPK(Long empresaId) {

**this**(empresaId, 0L);

}

**public** Long getCEmpresaId() {

**return** **this**.cEmpresaId;

}

**public** **void** setCEmpresaId(Long cEmpresaId) {

**this**.cEmpresaId = cEmpresaId;

}

**public** Long getCDepartamentoId() {

**return** **this**.cDepartamentoId;

}

**public** **void** setCDepartamentoId(Long cDepartamentoId) {

**this**.cDepartamentoId = cDepartamentoId;

}

**public** **boolean** equals(Object other) {

**if** (**this** == other) {

**return** **true**;

}

**if** (!(other **instanceof** DepartamentoPK)) {

**return** **false**;

}

DepartamentoPK castOther = (DepartamentoPK) other;

**return** (**this**.cEmpresaId == castOther.cEmpresaId)

&& (**this**.cDepartamentoId == castOther.cDepartamentoId);

}

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** hash = 17;

hash = hash \* prime + **this**.cEmpresaId.intValue();

hash = hash \* prime + **this**.cDepartamentoId.intValue();

**return** hash;

}

}

## @GeneratedValue

La anotación @GeneratedValue se utiliza para declarar un atributo ya anotado con @Id como valor generado (secuencia, autoincremental, etc, …). Precede a la declaración de un atributo de la clase y los atributos más utilizados son:

**generator** referencia al nombre de una anotación del tipo @SequenceGenerator o @TableGenerator que declara la estrategia de generación.

**strategy** define la estrategia de generación de forma directa. Se trata de un enumerado de los siguientes tipos: GenerationType.IDENTITY (autoincremental), GenerationType.TABLE (tabla intermedia con los IDs), GenerationType.SEQUENCE(secuencia SQL) y GenerationType.AUTO (la implementación decide la estrategia)

Ejemplo:

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

prívate Integer cEmpresaId;

## @SecuenceGenerator

La anotación @SecuenceGenerator se utiliza para declarar un generador de identidades basado en una secuencia de base de datos. Se puede declarar precediendo a la declaración de la clase o de cualquiera de sus atributos de la clase. Los atributos mas comunes son:

**name** nombre del generador para ser referenciado por @GeneratedValue

**allocationSize t**amaño del incremento de la secuencia

**initialValue v**alor inicial de la secuencia

**sequenceName** nombre de la secuencia

## @Column

La anotación @Column se utiliza para declarar el mapeo un atributo de una clase a una columna de una tabla de base de datos. Precede a la declaración de un atributo de una clase y los atributos más utilizados son:

**name** nombre de la columna

**table** nombre de la tabla. Se puede omitir si se ha utilizado la anotación @Table en la definición de la entidad.

**insertable** indica si la columna puede ser utilizada en operaciones de inserción

**length** tamaño de la columna. Solo aplicable a los tipos compatibles con tamaños

**nullable** indica si la columna admite el valor null.

**unique** indica si la columna admite valores duplicados

**updatable** indica si la columna puede ser utilizada en operaciones de actualización

## @JoinColumn

La anotación @Column se utiliza para declarar el mapeo un atributo de una clase a una columna de una tabla de base de datos que forma parte de una relación entre entidades. Normalmente esta anotación va acompañada de alguna de las anotaciones de multiplicidad (@OneToOne, o @ManyToOne) que detallan la relación entre las entidades. Precede a la declaración de un atributo de una clase y los atributos más utilizados son:

**name** nombre de la columna

**table** nombre de la tabla. Se puede omitir si se ha utilizado la anotación @Table en la definición de la entidad.

**insertable** indica si la columna puede ser utilizada en operaciones de inserción

**length** tamaño de la columna. Solo aplicable a los tipos compatibles con tamaños

**nullable** indica si la columna admite el valor null

**unique** indica si la columna admite valores duplicados

**updatable** indica si la columna puede ser utilizada en operaciones de actualización

**referencedColumnName** nombre de la columna referenciada en el otro extremo de la relación

## @JoinTable

## @ManyToOne

La anotación @ManyToOne sirve para definir el extremo \* en una relación \*–1. Precede a la declaración de un atributo de una clase y los atributos más utilizados son:

cascade tipos de acción en cascada en respuesta de acciones de inserción, actualización y borrado. Los valores permitidos son CascadeType.DETACH para desligado en cascada, CascadeType.MERGE para ligado en cascada, CascadeType.PERSIST para persistencia en cascada, CascadeType.REFRESH para refresco en cascada, CascadeType.REMOVE para borrado en cascada, CascadeType.ALL para todos los anteriores

fetch indica si el la relación del otro extremo se obtendrá al obtener la entidad y si se demorará al acceso al atributo. Los valores posibles son FetchType.EAGER para la primera opción y FetchType.LAZY para la segunda que es la opción por defecto

targetEntity Tipo de la clase Java (Class) del otro extremo de la relación

## @OneToOne

La anotación @OneToOne sirve para definir un extremo en una relación 1–\*. Precede a la declaración de un atributo de una clase y los atributos más utilizados son:

cascade tipos de acción en cascada en respuesta de acciones de inserción, actualización y borrado. Los valores permitidos son CascadeType.DETACH para desligado en cascada, CascadeType.MERGE para ligado en cascada, CascadeType.PERSIST para persistencia en cascada, CascadeType.REFRESH para refresco en cascada, CascadeType.REMOVE para borrado en cascada, CascadeType.ALL para todos los anteriores

fetch indica si el la relación del otro extremo se obtendrá al obtener la entidad y si se demorará al acceso al atributo. Los valores posibles son FetchType.EAGER para la primera opción y FetchType.LAZY para la segunda que es la opción por defecto

targetEntity Tipo de la clase Java (Class) del otro extremo de la relación

mappedBy nombre del atributo de la clase del otro extremo de la relación que referencia a este extremo.

## @OneToMany

La anotación @OneToMany sirve para definir el extremo 1 en una relación 1–\*. Precede a la declaración de un atributo de una clase y los atributos más utilizados son:

cascade tipos de acción en cascada en respuesta de acciones de inserción, actualización y borrado. Los valores permitidos son CascadeType.DETACH para desligado en cascada, CascadeType.MERGE para ligado en cascada, CascadeType.PERSIST para persistencia en cascada, CascadeType.REFRESH para refresco en cascada, CascadeType.REMOVE para borrado en cascada, CascadeType.ALL para todos los anteriores

fetch indica si el la relación del otro extremo se obtendrá al obtener la entidad y si se demorará al acceso al atributo. Los valores posibles son FetchType.EAGER para la primera opción y FetchType.LAZY para la segunda que es la opción por defecto

targetEntity Tipo de la clase Java (Class) del otro extremo de la relación

mappedBy nombre del atributo de la clase del otro extremo de la relación que referencia a este extremo.

## @ManyToMany

La anotación @ManyToMany sirve para definir unextremo en una relación \*–\*. Precede a la declaración de un atributo de una clase y los atributos más utilizados son:

cascade tipos de acción en cascada en respuesta de acciones de inserción, actualización y borrado. Los valores permitidos son CascadeType.DETACH para desligado en cascada, CascadeType.MERGE para ligado en cascada, CascadeType.PERSIST para persistencia en cascada, CascadeType.REFRESH para refresco en cascada, CascadeType.REMOVE para borrado en cascada, CascadeType.ALL para todos los anteriores

fetch indica si el la relación del otro extremo se obtendrá al obtener la entidad y si se demorará al acceso al atributo. Los valores posibles son FetchType.EAGER para la primera opción y FetchType.LAZY para la segunda que es la opción por defecto

targetEntity Tipo de la clase Java (Class) del otro extremo de la relación

mappedBy nombre del atributo de la clase del otro extremo de la relación que referencia a este extremo.

En una declaración \*––\* de este tipo en uno de los extremos de la relación se utilizará en atributo mappedBy mientras que en el otro se utilizará la anotación @JoinTable.

## @Temporal

## @NamedQuery

## @NamedQuerys

Anotación de una entidad

Vamos a realizar un ejemplo de anotación de una clase java para convertirla en una entidad persistente. Modelaremos la clase Empresa que modela los datos de una empresa. La clase Java sin anotar es la siguiente (solo se muestran los atributos omitiéndose los métodos:

**public** **class** Empresa **implements** Serializable {

**private** Long cEmpresaId;

**private** String dNombre;

**private** String cifNif;

**private** Date fCreacion;

**private** Date fModificacion;

**private** List<Departamento> departamentos = **new** ArrayList<Departamento>();

**…**

}

Comenzamos anotando la clase como entidad y mapeándola a una tabla:

@Entity

@Table(schema=”PRUEBAS”, name=”T\_EMPRESA”)

**public** **class** Empresa **implements** Serializable {

**…**

Luego mapeamos la clave primaria con @Id y @Column:

…

@Id

@Column(name=”C\_EMPRESA\_ID”)

**private** Long cEmpresaId;

…

Luego mapeamos el resto de atributos a columnas mediante @Column:

…

@Column(name=”D\_NOMBRE”)

**private** String dNombre;

@Column(name=”C\_CIF\_NIF”, unique=true)

**private** String cifNif;

@Column(name=”F\_CREACION”)

@Temporal(type=TemporalType.TIMESTAMP)

**private** Date fCreacion;

@Column(name=”F\_MODIFICACION”)

@Temporal(type=TemporalType.TIMESTAMP)

**private** Date fModificacion;

…

Por ahora no anotamos la relación entre Empresas y departamentos.

Hemos visto como anotar una clase para convertirla en una entidad. Necesitamos ahora configurar la conexión entre las entidades y la base de datos relacional en las que persisten.

Configuración de la persistencia

Para que un proyecto Java pueda utilizar persistencia JPA hay que realizar ciertas tareas de configuración en el mismo, independientemente de la implementación utilizada, mediante la creación de ficheros en nuestra estructura del proyecto. La configuración de extensiones dependientes de la implementación se realizará en esos mismos ficheros.

# Ficheros de configuración de persistencia

## El fichero persistence.xml

Este fichero contiene la información de configuración de las diferentes unidades de persistencia (PersistenceUnits). En cada unidad de persistencia definimos:

* Proveedor de persistencia.
* Tipo de transacción.
* Configuración del DataSource.
* Configuración JDBC para tipos de transacción RESOURCE\_LOCAL.
* Clases persistentes.
* Propiedades específicas del proveedor.

Ejemplo de fichero de persistencia:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<persistence version=*"1.0"* xmlns=*"http://java.sun.com/xml/ns/persistence"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/persistence*

*http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd"*>

<persistence-unit name=*"default"* transaction-type=*"RESOURCE\_LOCAL"*>

<class>curso.jpa.model.Empresa</class>

<class>curso.jpa.model.Departamento</class>

<class>curso.jpa.model.DepartamentoPK</class>

<class>curso.jpa.model.Cliente</class>

<class>curso.jpa.model.Proyecto</class>

<class>curso.jpa.model.Empleado</class>

<exclude-unlisted-classes>false</exclude-unlisted-classes>

<properties>

<property name=*"eclipselink.jdbc.batch-writing"* value=*"JDBC"* />

<property name=*"eclipselink.jdbc.url"*

value=*"jdbc:oracle:thin:@jcylbd5203.csbs.jcyl.es:1526:DECSBS10"* />

<property name=*"eclipselink.jdbc.user"* value=*"pruebas"* />

<property name=*"eclipselink.jdbc.password"* value=*"pruebas"* />

<property name=*"eclipselink.jdbc.driver"*

value=*"oracle.jdbc.driver.OracleDriver"* />

<property name=*"eclipselink.logging.level"* value=*"WARNING"* />

</properties>

</persistence-unit>

</persistence>

Vamos a ver los diferentes parámetros de configuración del fichero persistence.xml:

name Nombra la unidad de persistencia que estamos configurando

transaction-type Admite dos valores– RESOURCE\_LOCAL para definir los atributos de la conexión mediante properties y JTA para utilizar un Datasource ubicado en el contenedor.

class Declaramos todas las clases que se refieren a entidades

propiedades Definimos las propiedades dependientes de la implementación JPA.

jta-datasource Se utiliza cuando el valor de transaction-type es JPA. Declara un referencia jndi al Datasource del contenedor (p.e. jdbc/AppPoolDS).

JPQL

# JPQL BASICO

Como vimos en el capitulo anterior, con EntityManager estamos limitados a realizar consultas en la base de datos proporcionando la identidad de la entidad que deseamos obtener, y solo podemos obtener una entidad por cada consulta que realicemos. JPQL nos permite realizar consultas en base a multitud de criterios, como por ejemplo propiedades de la entidad almacenada o condiciones booleanas, y obtener más de un objeto por consulta. Veamos el ejemplo más simple posible:

SELECT e FROM Empresa e

La línea anterior obtiene todas las instancias de la clase Empresa desde la base de datos. La expresión puede parecer un poco extraña la primera vez que se ve, pero es muy sencilla. Las palabras SELECT y FROM tienen un significado similar a las sentencias homónimas de SQL, indicando que se quiere seleccionar (SELECT) cierta información desde (FROM) cierto lugar. La segunda p es un alias para la clase Empresa, y ese alias es usado por la primera e (llamada 'expresión') para acceder a la clase a la que refiere el alias o a sus propiedades. El siguiente ejemplo nos ayudara a comprender esto mejor:

SELECT e.dNombre FROM Empresa e

El alias e nos permite utilizar la expresión e.dNombre para obtener los títulos de todas las películas almacenadas en la base de datos. Las expresiones JPQL utilizan la notación de puntos, convirtiendo tediosas consultas en algo realmente simple:

SELECT c.propiedad.subPropiedad.subSubPropiedad FROM Clase c

JPQL también nos permite obtener resultados referentes a mas de una propiedad:

SELECT e.dNombre, e.cifNif FROM Empresa e

Todas las sentencias anteriores (que más tarde veremos cómo ejecutar) devuelven o un unico valor o un conjunto de ellos. Podemos eliminar los resultados duplicados mediante la clausula DISTINCT:

SELECT DISCTINCT e.dNombre FROM Empresa e

Asimismo, el resultado de una consulta puede ser el resultado de una función agregada aplicada a la expresión:

SELECT COUNT(e) FROM Empresa e

En el ejemplo anterior, COUNT() es una función agregada de JPQL que cuenta el numero de ocurrencias tras realizar la consulta. El valor devuelto por la función agregada es lo obtenemos. Otras funciones agregadas son AVG para la media aritmética, MAX para el valor máximo, MIN para el valor mínimo y SUM para la suma de todos los valores.

# SENTENCIAS CONDICIONALES

Ahora que ya sabemos cómo realizar consultas básicas, vamos a introducir conceptos más complejos (pero aun simples). El primero de ellos es el de consulta condicional, aplicado con la clausula WHERE, la cual restringe los resultados devueltos por una consulta en base a ciertos criterios lógicos:

SELECT e FROM Empresa e WHERE e.cifNif = ‘11111111H’

La sentencia anterior obtiene todas las instancias de Empresa almacenadas en la base de datos con un nif igual a 11111111H. Las sentencias condicionales pueden contener varias condiciones:

SELECT e FROM Empresa e WHERE e.cifNif == ‘11111111H’ AND e.dNombre = 'EMPRESA1'

La sentencia anterior obtiene todas las instancias de Empresa con un nif igual a 11111111H y cuya propiedad dNombre sea igual a EMPRESA1. La otra sentencia condicional además de AND es OR . Veamos más ejemplos:

SELECT p FROM Pelicula p WHERE p.duracion BETWEEN 90 AND 150

La sentencia anterior obtiene todas las instancias de Pelicula con una duración entre (BETWEEN) 90 y (AND) 150 minutos. BETWEEN puede ser convertido en NOT BETWEEN en cuyo caso se obtendrías todas las películas que una duración que no se encuentre dentro del margen 90-150 minutos. Otro operador de comparación muy útil es [NOT] LIKE (NOT es opcional, como en el ejemplo anterior), el cual nos permite comparar una cadena de texto con comodines con las propiedades de una entidad almacenada en la base de datos. Veamos un ejemplo para comprenderlo:

SELECT e FROM Empresa e WHERE e.dNombre LIKE ‘EMPRESA%'

La sentencia anterior obtiene todas las instancias de Empresa cuyo nombre sea como (LIKE) EMPRESA...', es decir, la palabra 'EMPRESA' seguida de mas caracteres (entre cero y varios). El comodín que representa 'cero o mas caracteres' es el símbolo de porcentaje (%). El otro comodín aceptado por LIKE es el carácter de barra baja (\_) el cual representa un solo caracter.

# PARAMETROS DINAMICOS

Podemos añadir parámetros dinámicamente a nuestras sentencias JPQL de dos maneras: por posición y por nombre. La sentencia a continuación indica parámetros por posición:

SELECT e FROM Empresa e WHERE e.dNombre = ?1

Y la siguiente, parámetros por nombre:

SELECT e FROM Empresa e WHERE e.dNombre = :titulo

En el momento de realizar la consulta, podemos pasar los parámetros a la sentencia JPQL. Esto lo veremos más adelante

# ORDENAR LOS RESULTADOS

Cuando realizamos una consulta a la base de datos, podemos ordenar los resultados mediante la clausula ORDER BY (ordenar por), la cual admite ordenamiento ascendente mediante la clausula ASC (comportamiento por defecto) u en orden descendiente mediante la clausula DESC:

SELECT e FROM Empresa e ORDER BY e.dNombre DESC

La sentencia anterior podría tener una clausula WHERE entre SELECT y ORDER BY para restringir los resultados devueltos. Además, puedes incluir múltiples expresiones de ordenación:

SELECT e FROM Empresa e ORDER BY e.dNombre DESC, e.cifNif ASC

# OPERACIONES DE ACTUALIZACION

JPQL puede realizar operaciones de actualización en la base de datos mediante la sentencia UPDATE:

UPDATE Articulo a SET a.descuento = 15 WHERE a.precio > 50

La sentencia anterior actualiza (UPDATE todas las instancias de Articulo con un precio mayor de 50 aplicándoles un descuento de 15.

# OPERACIONES DE BORRADO

De forma muy similar al punto anterior, JPQL puede realizar operaciones de borrado en la base de datos mediante la sentencia DELETE:

DELETE FROM Empresa e WHERE e.cEmpresaId > 190

La sentencia anterior elimina (DELETE) todas las instancias de Empresa cuyo id sea mayor de 190.

# CONSULTAS CON NOMBRE (ESTATICAS)

Las consultas con nombre son diferentes de las sentencias dinamicas que hemos visto hasta ahora en el sentido en que no pueden cambiar: son leidas y transformadas en sentencias SQL cuando el programa arranca en lugar de cada vez que son ejecutadas. Este comportamiento estático las hace más eficientes y por tanto ofrecen un mejor rendimiento. Las consultas con nombre son definidas mediante metadatos (recordemos que los metadatos se definen mediante anotaciones o configuración XML) definidos en las propias entidades. Veamos un ejemplo:

@Entity

@NamedQuery(name="buscarTodos", query="SELECT e FROM Empresa e")

public class Empresa { ... }

El código anterior define una consulta con nombre mediante la anotación @NamedQuery. Esta anotación necesita dos atributos: name que define el nombre de la consulta, y query que define la sentencia JPQL a ejecutar. El nombre de la consulta debe ser único dentro de su unidad de persistencia, de manera que no puede existir otra entidad definiendo una consulta con nombre que se llame buscarTodas. Para evitar que podamos modificar por error la sentencia, es una buena idea utilizar una constante definida dentro de la propia entidad como nombre de la consulta:

@Entity

@NamedQuery(name= BUSCAR\_TODAS, query="SELECT e FROM Empresa p")

public class Empresa {

public static final String BUSCAR\_TODAS = "Empresa.buscarTodas";

...

}

Tanto de la primera manera, como de esta ultima, una vez definida una consulta con nombre podemos ejecutarla desde nuestra aplicación mediante el método createNamedQuery():

Query query = em.createNamedQuery(Empresa.BUSCAR\_TODAS);

List<Empresa> resultados = query.getResultList();

createNamedQuery() requiere un parámetro de tipo String que contenga el nombre de la consulta (el cual hemos definido en @NamedQuery(name=...). Una vez obtenida la colección de resultados, podemos trabajar con ellos de la manera habitual.

# CONSULTAS NATIVAS SQL

El tercer y ultimo tipo de consultas que vamos a ver requiere una sentencia SQL en lugar de una JPQL:

String sql = "SELECT \* FROM T\_EMPRESAS";

Query query = em.createNativeQuery(sql);

Las consultas nativas SQL pueden ser definidas de manera estática como las consultas con nombre, obteniendo los mismos beneficios de eficiencia y rendimiento. Para ello, necesitamos utilizar de nuevo metadatos:

@Entity

@NamedNativeQuery(name=Empresa.BUSCAR\_TODAS, query="SELECT \* FROM T\_EMPRESAS")

public class Empresa {

public static final String BUSCAR\_TODAS = "Empresa.buscarTodas";

...

}

Tanto las consultas con nombre como las consultas nativas SQL estáticas son muy útiles para definir consultas inmutables (por ejemplo buscar todas las instancias de una entidad en la base de datos, como hemos hecho en los ejemplos anteriores), aquellas que se mantienen entre distintas ejecuciones del programa y que son usadas frecuentemente, etc.

Prácticas

# Configuración común a todas las prácticas

En todos los proyectos que se realicen para las prácticas necesitaremos:

1. Una base de datos con las relaciones a mapear.
2. Un fichero de configuración de persistencia persistence.xml

La base de datos que utilizaremos será una base de datos ubicada en un servidor Oracle 10g. Los datos de conexión son los siguientes:

**Host** jcylbd5203.csbs.jcyl.es

**Port** 1526

**SID** DECSBS10

**Url** jdbc:oracle:thin:@jcylbd5203.csbs.jcyl.es:1526:DECSBS10

**Usuario**  PRUEBAS

**Password**  PRUEBAS

Para las prácticas utilizaremos el siguiente fichero persistence.xml:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<persistence version=*"1.0"*

xmlns=*"http://java.sun.com/xml/ns/persistence"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/persistence*

*http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd"*>

<persistence-unit name=*"practica"* transaction-type=*"RESOURCE\_LOCAL"*>

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

<properties>

<property name=*"javax.persistence.transactionType"*

value=*"RESOURCE\_LOCAL"*></property>

<property name=*"javax.persistence.jdbc.url"*

value=*"* *jdbc:oracle:thin:@jcylbd5203.csbs.jcyl.es:1526:DECSBS10"*>

<property name=*"javax.persistence.jdbc.user"* value=*"pruebas"*>

<property name=*"javax.persistence.jdbc.password"* value=*"pruebas"*>

<property name=*"javax.persistence.jdbc.driver"*

value=*"oracle.jdbc.driver.OracleDriver"/*>

<property name=*"eclipselink.ddl-generation"* value=*"create-tables"* />

<property name=*"eclipselink.application-location"* value=*"C:\temp\curso"* />

<property name=*"eclipselink.create-ddl-jdbc-file-name"*

value=*"create.sql"* />

<property name=*"eclipselink.drop-ddl-jdbc-file-name"* value=*"drop.sql"* />

<property name=*"eclipselink.ddl-generation.output-mode"*

value=*"both"* />

<property name=*"eclipselink.logging.level"* value=*"WARNING"* />

</properties>

</persistence-unit>

</persistence>

Práctica 1

Modelado de un chat

El mapeo de atributos y relaciones de los objetos es la parte más importante del proceso de configuración de un proyecto para la persistencia de sus objetos.

Utilizaremos anotaciones Java5 para definir el mapeo de atributos a columnas de una tabla, la generación de claves primarias, el mapeo de relaciones 1–1, 1­–N y N–M entre objetos, la generación de secuencias.

Vamos a introducir un problema simple de ejemplo que nos servirá para el desarrollo de este ejemplo:

*Se quiere modelar la persistencia de un sistema de chat. En un chat pueden existir varias habitaciones con un nombre que identifica al tema que se trata en la habitación. Existen participantes registrados en el sistema que pueden entrar y salir de las habitaciones de chat. Cuando un participante se encuentra en una habitación puede enviar mensajes que serán recibidos por los demás participantes que se encuentren en ese momento en la habitación. En cualquier momento un participante puede enviar un mensaje privado a otro que se encuentre en la habitación y este responderle por la misma vía. Se deben de registrar todos los mensajes emitidos así como la presencia o no de un participante en una determinada habitación.*

Este problema se va a utilizar para ver los diferentes mecanismos de mapeo de campos y entidades.

El diagrama de clases con el modelo de análisis de este problema sería:

Figura : Modelo de análisis de la aplicación chat



Se observan las siguientes relaciones entre entidades:

1. Existen N participantes registrados en un determinado momento en una habitación. Relación 1:N.
2. Un mensaje es emitido por un participante en una habitación. Relaciones N:1 entre el mensaje y el participante y N:1 entre el mensaje y el participante.
3. Un mensaje privado es un mensaje. Relación 1:1 entre mensaje y mensaje privado. Para la herencia podemos:
   1. Crear una nueva tabla con la PK compartida como FK.
   2. Crear una nueva columna en la entidad mensaje con el participante de destino y otra columna para identificar el tipo de mensaje (privado o no)
4. Un mensaje privado es para un participante. Relación N:1.

Una vez pasado a modelo relacional tendríamos el siguiente diagrama:

Figura : Modelo relacional del esquema CHAT



# Generación de las clases

Veamos como quedaría cada una de las clases. Empecemos con la clase *Habitacion*.

Listado ‑: Clase Habitacion

@Entity

@Table(name = "CHAT\_HABITACIONES")

public class Habitacion {

@Id

@Column(name = "C\_HABITACION\_ID", length = 10)

private Long id;

@Column(name = "D\_NOMBRE", length = 100, nullable = false, unique = true)

private String nombre;

...

*Aquí irian los métodos get y set de los atributos*

...

}

La primera anotación que nos encontramos es *@Entity*. Con esto indicamos a la implementación JPA que esta clase debe de ser gestionada. Aparte de esta indicación debemos registrar la clase en el fichero *persistence.xml* cuando la registremos en alguna unidad de persistencia.

La siguiente anotación que nos encontramos es *@Table*. Acepta como parámetro el atributo *name* con el que indicamos el nombre de la tabla donde se realizará el mapeo de atributos. En este caso indicamos que la clase se mapeará a la tabla *CHAT\_HABITACION*.

La siguientes anotaciones se aplican al atributo *id* de la clase. Son *@id* y *@Column.* Con *@Id* declaramos que este atributo será *primary key* de la relación y con el atributo *@Column* declaramos que el atributo se mapeará a la columna *C\_HABITACION\_ID* de la tabla. En esta anotación establecemos que la longitud es de 10 mediante el atributo *length*.

Para el atributo nombre repetimos la anotación *@Column* con nuevos atributos: *nullable* y *unique*. Con esto especificamos que la columna de la tabla es *NOT NULL* y *UNIQUE*.

Con esto es suficiente para poder dar persistencia y recuperar objeto *Habitacion*.

La clase Participante quedaría de igual forma:

Listado ‑: Clase Participante

@Entity

@Table(name = "CHAT\_PARTICIPANTES")

public class Participante implements Serializable{

@Id

@Column(name = "C\_PARTICIPANTE\_ID", length = 10)

private Long id;

@Column(name = "D\_NICK", length = 100, nullable = false, unique = true)

private String nick;

...

*Aquí irian los métodos get y set de los atributos*

...

}

La clase Mensaje tiene más contenido:

Listado ‑: Clase Mensaje

@Entity

@Table(name = "CHAT\_MENSAJES")

public class Mensaje {

@Id

@JoinColumn(name="C\_PRODUCTOR\_ID")

protected Participante productor;

@Id

@Column(name = "C\_MENSAJE\_ID", length = 10)

private Long id;

@Column(name = "D\_MENSAJE", length = 500, nullable = false)

protected String mensaje;

@JoinColumn(name="C\_HABITACION\_ID")

protected Habitacion habitacion;

...

*Aquí irian los métodos get y set de los atributos*

...

}

Aparecen dos anotaciones *@Id* indicando que la *primary key* de la tabla tiene dos columnas, la columna *C\_MENSAJE\_ID* y la columna *C\_PRODUCTOR\_ID.* Además aparece la anotación *@JoinColumn* sobre el atributo productor de tipo *Participante*. Con esto estamos indicando que además de ser *primary key* es *foreugn key*. El gestor de persistencia deducirá viendo el atributo @Id de la clase Participante que debe utilizar la columna *C\_PRODUCTOR\_ID* como parte de la *FK* que apunta a la *PK* *C\_PARTICIPANTE\_ID* de la tabla *CHAT\_PARTICIPANTES*.

Falta por modelar la persistencia de la clase *MensajePrivado*. Al ser una especialización de la clase *Mensaje* (herencia) quedaría:

Listado ‑: Clase MensajePrivado

@Entity

public class MensajePrivado extends Mensaje {

@JoinColumn(name = "C\_DESTINATARIO\_ID")

private Participante destinatario;

...

*Aquí irian los métodos get y set de los atributos*

...

}

Tan solo modelamos el atributo *destinatario* correspondiente al *Participante* que recibe el mensaje. El resto de atributos se heredan de la clase *Participante*.

El mecanismo de herencia de entidades de JPA es bastante flexible. No hemos indicado nada acerca de la herencia salvo *extends*. Con esto JPA conoce lo suficiente para gestionar la especialización de entidades. Al no especificar la forma de extensión, JPA por defecto utiliza la misma tabla que *Mensaje* ampliándola con dos columnas más. Una es la columna por defecto *DTYPE*  que JPA utilizará para discriminar entre Mensaje y MensajePrivado, y otra es la *C\_DESTINATARIO\_ID* donde al igual que en *Mensaje* hará el join con Participante.

# Persistencia de objetos

Hasta ahora hemos anotado las clases para que estas puedan ser gestionadas por JPA. Ahora vamos a utilizar JPA para crear y recuperar objetos de la base de datos.

El proceso de acceso a la base de datos para recuperar objetos se realiza en 5 pasos:

1. Obtención del objeto *EntityManagerFactory*.
2. Creación de un objeto *EntityManager*.
3. Operaciones de persistencia (inserción, borrado, consulta).
4. Liberación del objeto *EntityManager*.
5. Liberación del objeto *EntityManagerFactory*.

Vamos a verlo en detalle.

## Obtención del objeto EntityManagerFactory

Es el primer paso a realizar. Se realiza de la siguiente forma:

EntityManagerFactory emf =

Persistence.createEntityManagerFactory(persistenceUnit);

Con esto conseguimos el acceso a la factoria de *EntityManagers*. En este momento se inicia la sincronización de los metadatos (anotaciones) con la base de datos. Esta operación puede y debe hacerse una sola vez. Al finalizar la aplicación debemos cerrar la factoría (paso 5).

## Creación de un objeto EntityManager.

Una vez obtenida la factoría debemos obtener un EntityManager. El EntityManager sigue la pista de todos los objetos persistentes durante la vida del EntityManager. Lo haremos con el siguiente código:

EntityManager em = emf.createEntityManager(“chat”);

Se le pasa como parámetro el nombre de la unidad de persistencia configurada en el fichero persistente.xml.

## Operaciones de persistencia (inserción, borrado, consulta).

Aquí debemos distinguir entre operaciones de consulta y el resto. Para las operaciones de consulta no se requiere iniciar una transacción aunque se puede hacer. Para el resto (*insert, update, delete*) es necesario iniciar una transacción antes de ejecutar operaciones de este tipo.

Para iniciar una transacción hacemos:

em.getTransaction().begin();

Y para finalizarla

em.getTransaction().commit();

o

em.getTransaction().rollback();

Veamos las diferentes operaciones:

Inserción

Si tenemos un nuevo objeto y queremos darlo persistencia, lo único que tenemos que hacer es:

Participante p = new Participante();

p.setId(1L);

p.setNombre(“HABITACION 1”);

em.persist(p);

Al ejecutar el commit se realizará la *insert* en la base de datos.

Borrado

Es similar al caso anterior:

Participante p = …

em.remove(p)

Al ejecutar el *commit* se realizará el *delete* en la base de datos

Actualización

La actualización es incluso más sencilla. Si partimos de un objeto p recuperado de la base de datos:

Participante p = …

p.setNombre(“NuevoNombre”);

Al ejecutar el *commit* se realizará el *update* en la base de datos.

Consulta

La recuperación de objetos persistentes de la base de datos se puede hacer de varias formas.

1. Recuperación por clave privada

Participante p = em.find(Participante.class, 23L);

Con esto indicamos al EntityManager que instancia un objeto de tipo participante con clave primaria Long 23.

1. Recuperación mediante lenguaje JPQL

Esto se verá en el capítulo 5.

1. Recuperación por acceso a un atributo persistente de una clase persistente

Cuando tenemos un objeto persistente (gestionado por un EntityManager), con solo acceder al atributo que sea a su vez un objeto persitente el EntityManager recuperará de la base de datos el objeto. Por defecto al cargar un objeto no se cargan sus atributos que no sean tipos básicos (String, Date, Long, Integer, …) sino que se cargan durante el acceso al atributo. Esto se puede configurar para que no sea así y el gestor los recupere en la recuperación del objeto que los contiene.

Mensaje m = em.find(Mensaje.class, ...);

Participante p = m.getParticipante(); // Aquí el gestor recupera de la base de datos el objeto participante asociado

## Liberación del objeto EntityManager.

Se realiza con la llamada al método *close()* del *EntityManager*.

em.close()

## Liberación del objeto EntityManagerFactory

Se realiza con la llamada al método *close()* del *EntityManager*.

Como ejemplo de lo anterior sería:

Listado ‑: Ejemplo de operación con objetos persistentes

...

**// Paso 1**

EntityManagerFactory emf = Persistence.getEntityManagerFactory(“chat”);

**// Paso 2**

EntityManager em = em.createEntityManager();

try {

**// Paso 3**

em.getTransaction.begin();

Participante p = new Participante();

p.setId(23L);

p.setNombre(“Mi habitacion”);

em.persist(p);

em.getTransaction().commit();

} finally {

if (em.getTransaction().isActive())

em.getTransaction().rollback();

**// Paso 4**

em.close();

**// Paso 5**

emf.close();

}

...

Modelado de una empresa

Modelar en entidades Java las tablas del siguiente diagrama:

