# Procesos de la Ingeniería Software
# Tema 4

## *Soporte Java para construcción de aplicaciones empresariales*

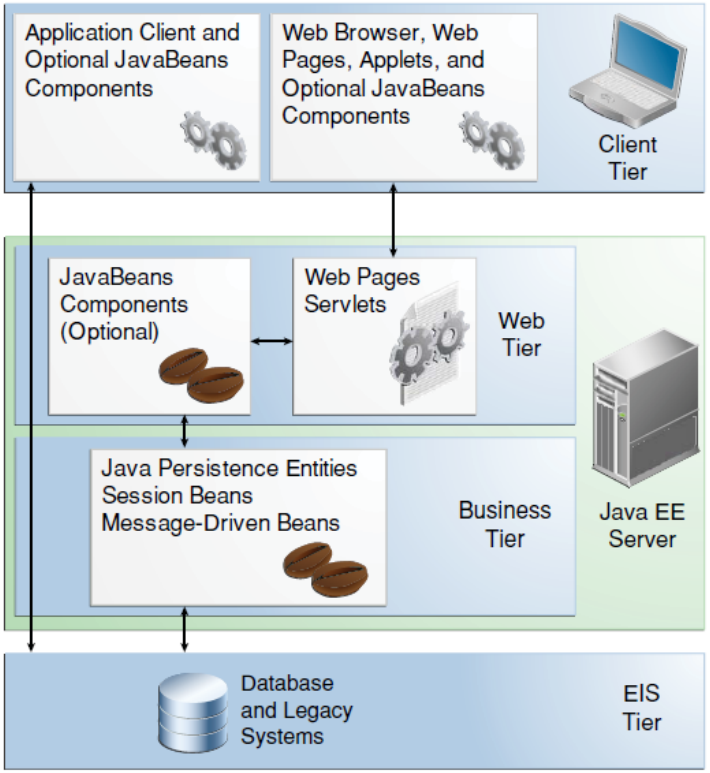
## *3. Capa de persistencia en Java EE: Java Persistence API*

# Bibliografía

- Lectura obligada
  - Antonio Goncalvez (2013): Beginning Java EE 7, Apress
    - Capítulo 4 - 6
- Lectura complementaria
  - Erik Jendrock et al. (2014): The Java EE 7 Tutorial
    - Capítulos 37 – 39

# Capa de persistencia en aplicaciones empresariales

- La capa de negocio de una aplicación empresarial modifica, actualiza y gestiona datos persistentes
- La capa de persistencia se encarga de realizar el enlace entre la capa de negocio y la BBDD
- En sistemas OO, la capa de persistencia se encarga de realizar el **Mapeado Objeto-Relacional** (ORM)
  - Mapeo de objetos software a relaciones o tuplas del modelo relacional
    - Para su persistencia y posterior recuperación
- Formas de implementar la capa de persistencia:
  - Con interfaces de bajo nivel: JDBC
  - A través de frameworks ORM: Ibatis, Hibernate, etc.



# Capa de persistencia en Java EE: Java Persistence API

- La implementación estándar de la capa de persistencia en Java EE es a través de JPA

- JPA (**Java Persistence API**) es una especificación para la implementación de ORM
  - Independiente del framework ORM o del gestor de la BBDD
    - JPA simplifica el ORM, pero quien realmente realiza el mapeo es el framework subyacente (proveedor de persistencia)
  - No es exclusiva de Java EE, se puede usar también en aplicaciones Java SE
  - Versión actual: 2.1
  - Paquete `javax.persistence`

- Los elementos principales de JPA son:
  - **Entidades** (Entity), con sus correspondientes metadatos (anotaciones o descriptores XML)
  - Gestor de entidades (**EntityManager**)
  - Lenguaje de consulta **JPQL** (Java Persistence Query Language)

- Implementación de referencia:
  - EclipseLink 2.0 (Open Source)

# Elementos de JPA: Entidades (Entity)

- Una **entidad** es un objeto de una aplicación cuyo estado se hace **persistente**
  - El estado lo forman los atributos (campos) o las propiedades (get/set) del objeto

- Una clase entidad representa una tabla en una BBDD relacional
  - Cada instancia (objeto) de la entidad representa una fila de dicha tabla
  - JPA sincroniza la clase entidad y la tabla de la BBDD
    - Gracias a metadatos en forma de anotaciones o descriptores xml
  - Cuando la entidad y la tabla están sincronizadas (más adelante veremos cómo) el contenedor se encarga de mapear las invocaciones en el objeto entidad (creación, modificación de atributos, etc.) a las operaciones en la BBDD

| Modelo Relacional (BBDD) | JPA |
|---|---|
| Relación (Tabla) | Clase entidad |
| Atributo (Columna) | Atributo/Propiedad de la clase |
| Tupla (Fila) | Instancia de la clase entidad |

# Entidades: Características principales

- Para que JPA reconozca una clase como entidad, la clase debe ser un POJO:
  - Anotado como **@Entity**
  - Con, al menos, un constructor público sin parámetros
    - Puede tener más constructores de utilidad
  - Con patrón getter/setter para aquellos atributos / propiedades que se quieran hacer persistentes
  - Ni la clase ni ninguno de los atributos/propiedades persistentes pueden ser final ni static
  - Si la entidad se va a usar como parámetro de una interfaz de negocio EJB, debe implementar la interfaz Serializable
    - Ponerlo siempre por defecto

- Reglas por defecto de mapeado de una entidad:
  - La clase se mapea a una tabla con el mismo nombre
  - Cada atributo/propiedad simple se mapea a una columna de la tabla con el mismo nombre y tipo y nullable = true
  - Para los atributos/propiedades múltiples, hay reglas más complejas (más adelante)

```
import javax.persistence.Entity;
import java.io.Serializable;

@Entity
public class Usuario implements Serializable {


   private String nombreUsuario;

   private String fechaAlta;

   public void Usuario() {
   }

   public void Usuario(String n) {
      nombre = n;
   }

   public String getNombreUsuario () {
      return nombreUsuario;
   }

   public void setNombreUsuario (String nombre) {
      this.nombreUsuario = nombre;
   }

   public String getEmail() {
      return nombreUsuario+"@ejemplo.com"
   }

   public void setEmail(String email) {
   }
}
```

# Elementos de JPA: Gestor de entidades (Entity Manager)

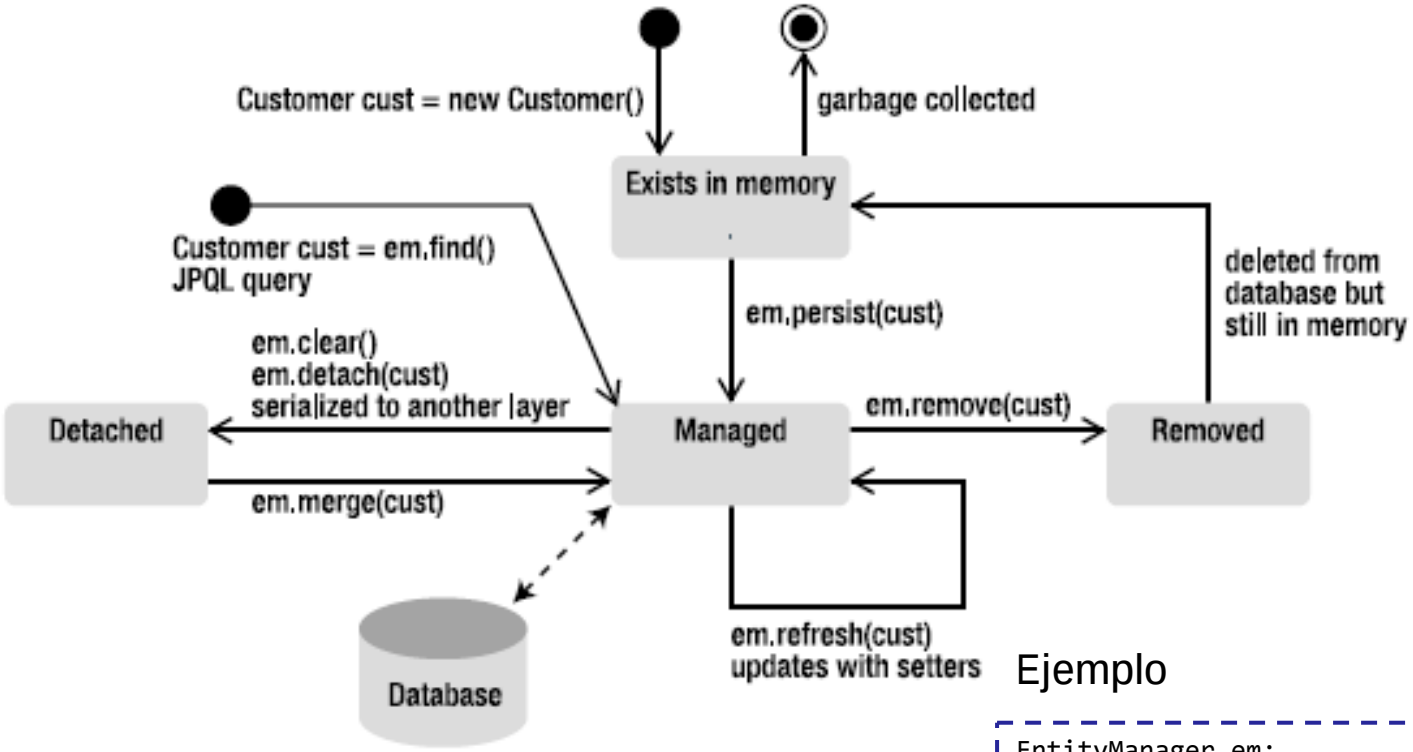
- El gestor de entidades, i.e. el gestor de persistencia en Java EE es el **EntityManager**
  - Proporciona la API que implementa el ORM:
    - Crear, actualizar, eliminar y acceder a entidades (operaciones **CRUD**)
    - Hacer **búsquedas** selectivas de entidades (a través de lenguaje JPQL)
  - Los métodos proporcionados trabajan al nivel de abstracción de OO
    - Hace transparente las invocaciones SQL o JDBC

- El EntityManager gestiona el ciclo de vida de las entidades
  - Una entidad es un simple POJO hasta el momento en que es asociado al EntityManager
    - Importante: **Las entidades SÍ se construyen con new** (a diferencia de los EJBs)

A partir de aquí el objeto nuevo es un objeto entidad, se dice que es un "**Managed Object**"
(En realidad el persist se ejecuta cuando finaliza la transacción en la que se ejecuta)

```
@Stateless
public class PropietariosDAO implements Serializable {

    private EntityManager em;

    public void anhadePropietario(Propietario nuevo) {

        // Hacemos el propietario persistente
        em.persist(nuevo);
    }
}
```

# Entity Manager: Ciclo de vida de entidades




Ejemplo

```
EntityManager em;
Propietario p1 = new Propietario(“11111111X”, “Pepe”);
Propietario p2 = new Propietario(“22222222X”, “Juan”);
em.persist(p1);
em.persist(p2);
p1.setNombre(“Juan”);
em.detach(p1);
p1.setNombre(“Manolo”);
P2.setNombre(“Andrés”);
System.out.println(p1.getNombre());
System.out.println(p2.getNombre());
```

# EntityManager API: Métodos principales

```
public interface EntityManager {

   /* Hace persistente (crea la tupla en la BBDD) y
    "managed" la entidad que se pasa como parámetro
    */
   public void persist(Object entity);

   /* Elimina la entidad del contexto de
    persistencia y de la BBDD */
   public void remove(Object entity);

   /* Actualiza en la BBDD y hace "managed" la
    entidad que se pasa como parámetro (que está
    detached). Si no existe la crea (como persist).
    En ambos casos retorna la entidad "managed"*/
   public <T> T merge(T entity);

   /* Actualiza la entidad que se pasa como
    parámetro con los valores que se encuentren en
    la BBDD*/
   public void refresh(Object entity);

   /* Desliga la entidad del contexto de
    persistencia (pasa de "managed" a "detached") */
   public void detach(Object entity);

   /* Comprueba si la entidad forma parte del
    contexto de persistencia (es "managed") */
   public boolean contains (Object entity);

   /* Sincronización explícita con la BBDD
    (modifica la BBDD con los valores actuales
    de todas las entities managed */
   public void flush();

   /* Búsqueda de entidades por clave primaria.
    La entidad que se retorna está "managed"
    Si no la encuentra retorna null */
   public <T> T find(Class<T> entityClass,
          Object primaryKey);

. . .

}
```

# Entidades: Anotaciones @Id y @Transient

- Toda clase entidad debe tener al menos un atributo que sirva como identificador único de cada instancia de la entidad
  - Anotado como **@Id**
  - Representa la primaryKey en la BBDD
  - La asignación de la anotación @Id define el criterio (atributos / propiedades) que se usa para definir el estado persistente de la entidad
    - Si se asigna a un atributo => El estado lo forman solo los valores de todos los atributos
    - Si se asigna a un par get/set => El estado lo forman solo los valores de las propiedades
- El estado persistente de la entidad lo forman los valores de todos sus atributos/propiedades excepto aquellos anotados con **@Transient**

```
import javax.persistence.Entity;
import java.io.Serializable;

@Entity
public class Usuario implements Serializable {

   @Id
   private String nombreUsuario;

   @Transient
   private String fechaAlta;

   public void Usuario() {
   }

   public void Usuario(String n) {
      nombre = n;
   }

   public String getNombreUsuario () {
      return nombreUsuario;
   }

   public void setNombreUsuario (String nombre) {
      this.nombreUsuario = nombre;
   }

   public String getEmail() {
      return nombreUsuario+"@ejemplo.com"
   }

   public void setEmail(String email) {
   }
}
```

# Entidades: Anotaciones

| Ámbito | Anotación (params) | Uso | Objetivo |
|---|---|---|---|
| Clase | @Entity | Obligatorio | Identifica clases entidad (las que se van a hacer persistentes) |
| | @Table (name, ...) | Opcional | Configura características de la tabla de la BBDD a la que se mapea la entidad, e.g. el nombre de la tabla. Valor por defecto: Nombre de la clase |
| Atributo/ Propiedad | @Id | Obligatorio | Identifica el atributo que funciona como primaryKey del objeto |
| | @GeneratedValue (value) | Opcional | Define el modo en que se genera la primaryKey (sólo para atributos @Id).<br>Valor por defecto: GenerationType.AUTO (el proveedor de persistencia decide la estrategia) |
| | @Transient | Opcional | Indica que el atributo/propiedad no se hace persistente |
| | @Column (name, type, nullable, length, ... ) | Opcional | Configura cómo se mapea el atributo/propiedad a la columna de la BBDD<br>Valores por defecto: Mismo nombre/tipo que el atributo, nullable = true y length = 255 (Strings) |
| | @Temporal (value) | Opcional | Define el modo en que se mapean atributos de tipo Date/Calendar. Posibles valores:<br>TemporalType.TIME, TemporalType.DATE, TemporalType.TIMESTAMP |
| | @Enumerated (value) | Opcional | Configura el modo en que se mapea un valor enumerado. Posibles Valores:<br>EnumType.STRING / EnumType.ORDINAL (Mejor usar el valor STRING) |
| | @JoinColumn (name, ...) | Opcional | Indica que un atributo actúa como clave foránea de otra entidad<br>name es el nombre de la columna de la clave foránea en la otra entidad |
| | @OneToOne (fetch, cascade, ...) | Opcional | Indica que el atributo está asociado con una tupla de otra Entidad |
| | @OneToMany (fetch, cascade, ...) | Opcional | Indica que el atributo está asociado con varias tuplas de otra Entidad |
| | @ManyToOne (fetch, cascade, ...) | Opcional | Indica que el atributo (lado N) está asociado con una tupla de otra Entidad |
| | @ManyToMany (fetch, cascade, ...) | Opcional | Indica que el atributo tiene una relación N:M con tuplas de otra Entidad |

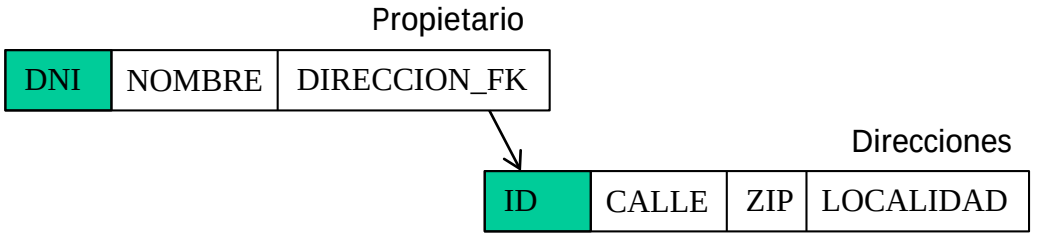
# Entidades: Relaciones OneToOne Unidireccional

```java
@Entity
public class Propietario implements Serializable {

    @Id
    private String DNI;

    private String nombre;

    @OneToOne
    @JoinColumn(name="direccion_fk")
    private Direccion direccion;
    // name = "DIRECCION_ID" por defecto
    // name = <Nombre atributo>_<PrimarykeyClaseRef>

    public void Propietario() {  }
    …
}
```

```java
@Entity
@Table(name="Direcciones")
public class Direccion implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    private String calle;

    @Column(name="zip")
    private String codigoPostal;

    private String localidad;

    public void Direccion() {

    }
    …
}
```

@OneToOne, @Column, etc. no son necesarias si no se modifica ningún valor por defecto

@JoinColumn en relaciones @OneToOne y @ManyToOne se mapea a una columna "FK" en la tabla de la propia clase dónde aparece la anotación

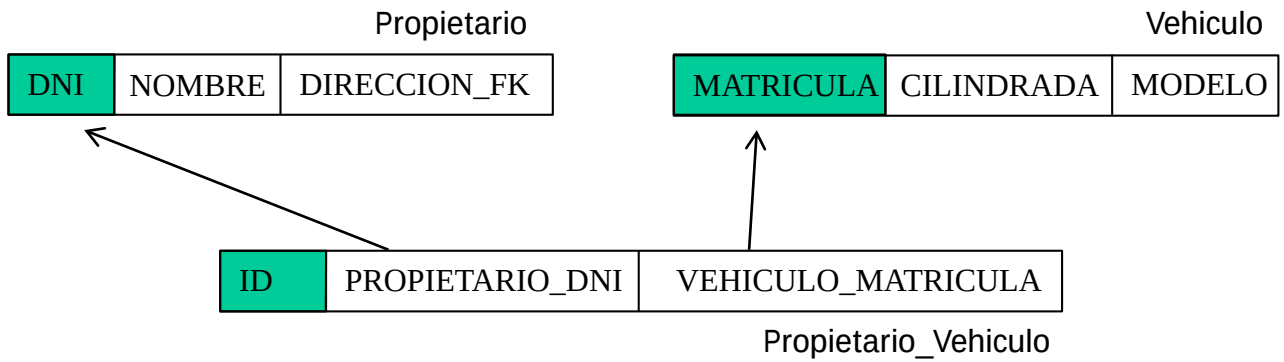


# Entidades: Relaciones OneToMany Unidireccional

```
@Entity
public class Propietario  implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   private List<Vehiculo> vehiculos;

   public void Propietario() {

   }
   …
}
```

```
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   public void Vehiculo() {

   }
   …
}
```

El mapeado por defecto de una asociación OneToMany es a través de una JoinTable



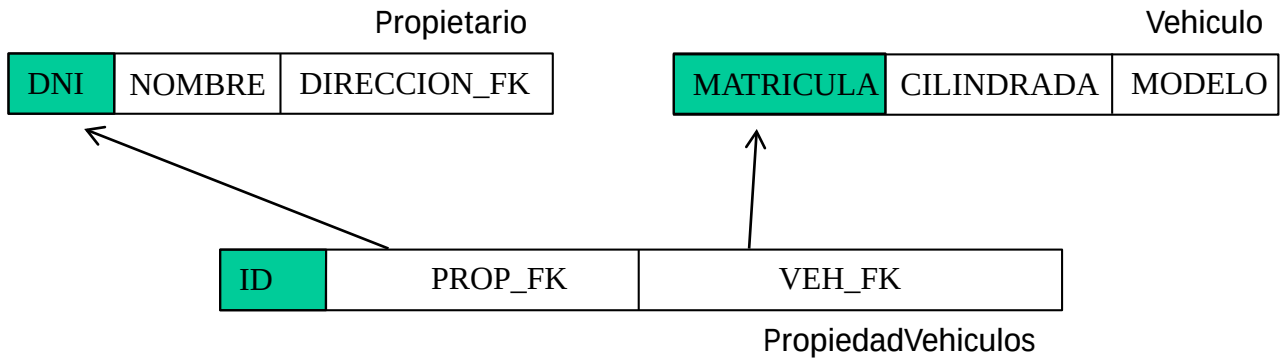
# Entidades: Relaciones OneToMany Unidireccional

```
@Entity
public class Propietario  implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   @OneToMany
   @JoinTable(name="PropiedadVehiculos",
              joinColumns=@JoinColumn(name="Prop_FK"),
              inverseJoinColumns=@JoinColumn(name="VEH_FK"))
   private List<Vehiculo> vehiculos;

   public void Propietario() {   }
   …
}
```

```
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   public void Vehiculo() {

   }
   …
}
```

Con @JoinTable podemos modificar la JoinTable generada

Se asigna en el lado "owner"

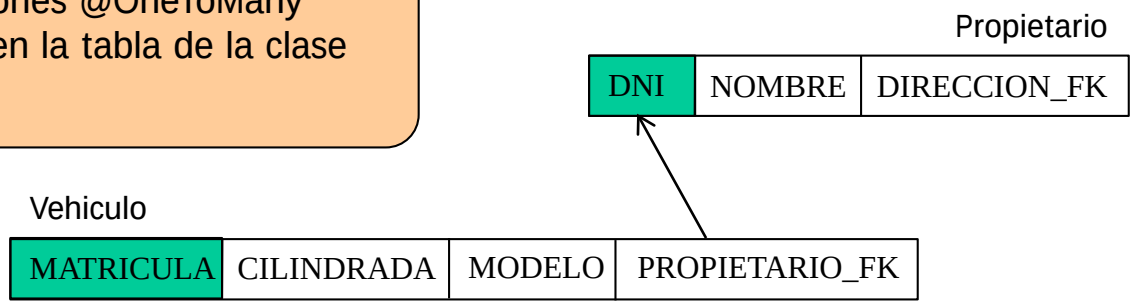


# Entidades: Relaciones OneToMany Unidireccional

```
@Entity
public class Propietario  implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   @OneToMany
   @JoinColumn(name="propietario_fk")
   private List<Vehiculo> vehiculos;

   public void Propietario() {

   }
   …
}
```

```
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   public void Vehiculo() {

   }
   …
}
```

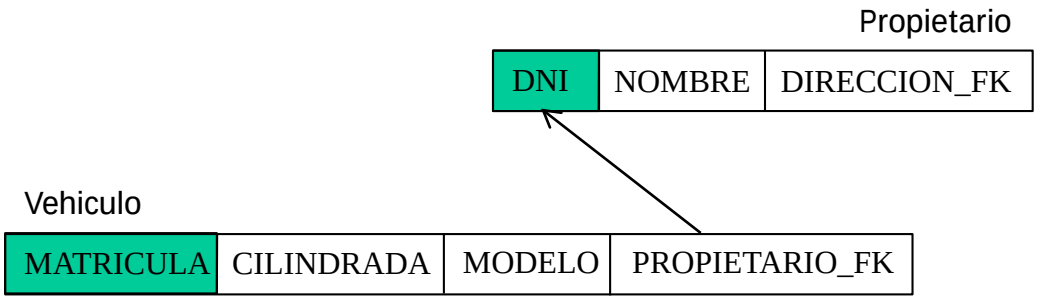
@JoinColum en relaciones @OneToMany se mapea a columna en la tabla de la clase referenciada



# Entidades: Relaciones bidireccionales

```java
@Entity
public class Propietario implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   // Lado inverso de la relación
   @OneToMany(mappedBy="propietario")
   private List<Vehiculo> vehiculos;

   public void Propietario() {

   }
   …
}
```

```java
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   // Lado poseedor (owner) de la relación
   @ManyToOne
   @JoinColumn(name="propietario_fk")
   private Propietario propietario;

   public void Vehiculo() {

   }

}
```




El poseedor (owner) de la relación es quién define cómo se realiza el mapping (con @JoinColum o @JoinTable)

En caso de JoinColumn, la columna con la foreign key se genera en la tabla del owner

El owner es el que NO tiene el atributo mappedBy (mappedBy indica quién posee la relación)

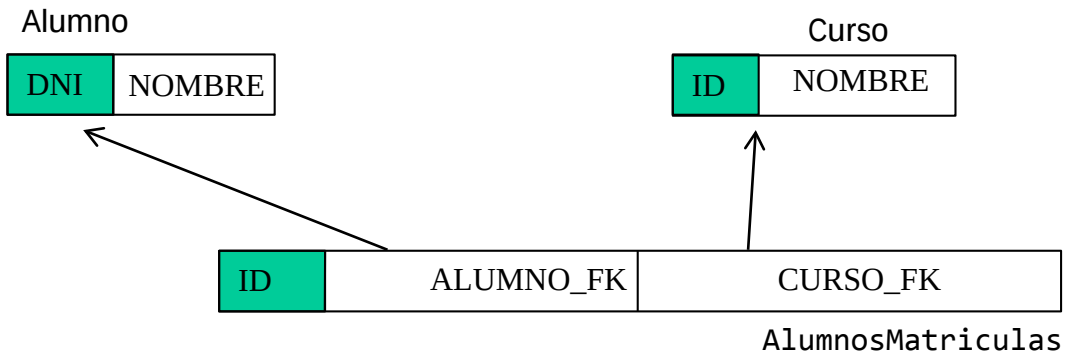
Decidir quién es el owner queda a decisión del diseñador, excepto en el caso de relaciones OneToMany(ManyToOne), donde JPA obliga a que el owner sea el lado Many.

# Entidades: Relaciones ManyToMany

```
@Entity
public class Curso implements Serializable {

   @Id
   private Long ID;
   private String nombre;

   @ManyToMany(mappedBy="cursos")
   private List<Alumno> alumnos;

   public void Curso() {

   }
   …
}
```

```
@Entity
public class Alumno implements Serializable {

   @Id
   private String dni;
   private String nombre;

   @ManyToMany
   @JoinTable(name="AlumnosMatriculas",
              joinColumns=@JoinColumn(name="Alumno_FK"),
              inverseJoinColumns=@JoinColumn(name="Curso_FK"))

   private List<Curso> cursos;

   public void Vehiculo() {

   }
   …
}
```

Las relaciones @ManyToMany se mapean siempre a JoinTable



# Entidades: Carga de entidades asociadas

- El atributo **fetch** de las anotaciones @OneToOne, @OneToMany, etc. define el modo en que se cargan en memoria las entidades referenciadas

- Posibles valores:
  - **FETCH.EAGER**: Carga inmediata
    - Puede sobrecargar el sistema
    - Un único acceso a la base de datos
  - **FETCH.LAZY**: Carga retardada
    - No sobrecarga el sistema
    - Muchas peticiones a la base de datos

- Valores por defecto:
  - @OneToOne => EAGER
  - @ManyToOne => EAGER
  - @OneToMany => LAZY
  - @ManyToMany => LAZY

# Entidades: Propagación de operaciones

- El atributo **cascade** de las anotaciones @OneToOne, @OneToMany, etc. define el modo en que las operaciones invocadas en la entidad principal se propagan a sus entidades relacionadas
  - El valor del atributo define cuáles de las operaciones se realizan en cascada
  - Posibles valores: ALL, PERSIST, MERGE, REMOVE, REFRESH
  - Valor por defecto: Ninguna operación

```
@Entity
public class Propietario implements
Serializable {

   ...
   @OneToOne
   private Direccion direccion;

   }
}
```

```
Propietario p= new Propietario("72111111", "Pepe");
Direccion d = new Direccion("Avda. de los Castros",
                            "39006", "Santander");
em.persist(p);
em.persist(d);

p.setDireccion(d);
```

```
@Entity
public class Propietario implements
Serializable {

   ...
   @OneToOne(cascade = Cascade_Type.PERSIST)
   private Direccion direccion;

   }
}
```

```
Propietario p= new Propietario("72111111", "Pepe");
Direccion d = new Direccion("Avda. de los Castros",
                            "39006", "Santander");
p.setDireccion(d);

em.persist(p);
```

# Entidades: Objetos Embeddable

- Un objeto **Embeddable** es aquel que no tiene una entidad persistente de por sí, sino que está contenido en otra entidad
  - Mismas reglas que una Entity, pero anotadas como `@Embeddable`
  - Se mapean a la misma tabla que la entidad contenedora
  - Relación de contención estricta
    - Si desaparece el contenedor, desaparece el contenido

```
@Entity
public class Propietario implements Serializable {

   @Id
   private String DNI;

   private String nombre;

   @Embedded
   private Direccion direccion;

   …

}
```

```
@Embeddable
public class Direccion implements Serializable {

   private String calle;

   @Column(name="zip")
   private String codigoPostal;

   private String localidad;

   …
}
```

Propietario

| DNI | NOMBRE | CALLE | ZIP | LOCALIDAD |
|---|---|---|---|---|

# Entity Manager: Unidad y Contexto de persistencia

- Un EntityManager mapea una **unidad de persistencia** (Persistence Unit) a una BBDD

- Unidad de persistencia: Conjunto de clases entidad
  - Tiene un nombre que la identifica unívocamente
  - La unidad de persistencia debe estar desplegada antes de poder hacer uso de ella (instrucciones para el despliegue en la siguiente transparencia)

- En ejecución, un EntityManager gestiona un **contexto de persistencia** (Persistence Context)
  - Gestionar = Mantener sincronización con la BBDD

- Contexto de persistencia: Conjunto de instancias de clases entidad, pertenecientes a una unidad de persistencia dada, gestionadas por el EntityManager en un determinado momento durante una transacción
  - Sólo “managed objects”
  - Si el contexto de persistencia se cierra, las instancias dejan de ser “managed”, es decir, dejan de estar sincronizadas con la BBDD
  - Al final de la transacción, todos las instancias entidad son actualizadas a la base de datos (flush implícito)

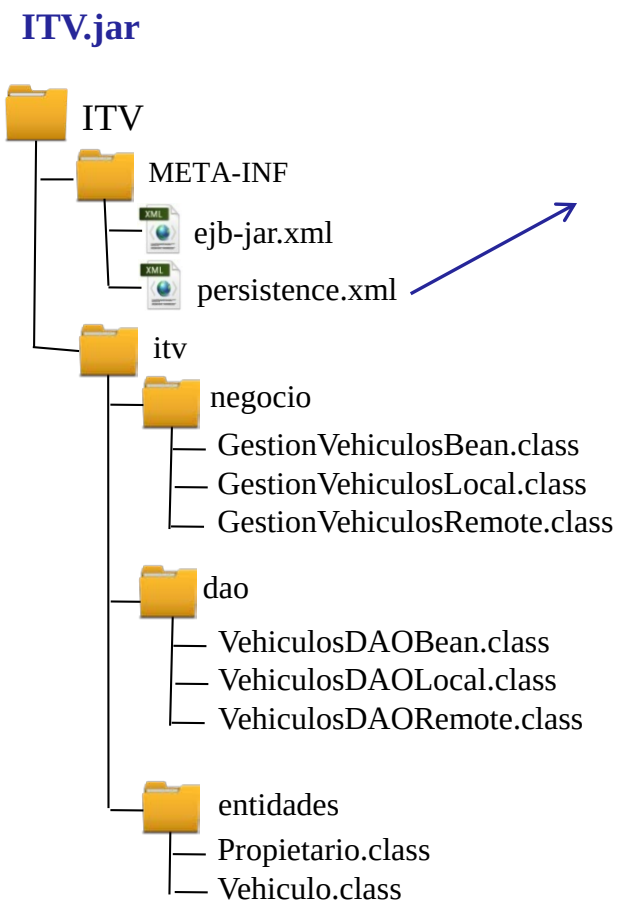
# Entity Manager: Despliegue de unidades de persistencia

- Para su despliegue, cada **unidad de persistencia** (o varias a la vez) se empaqueta dentro de un archivo .war o .jar

- Una unidad de persistencia contiene:
  - Las clases entidad que comprende
  - Un descriptor denominado **persistence.xml**
    - Se almacena en el directorio META-INF de la denominada raíz de la unidad de persistencia, que puede ser:
      - Archivo EJB JAR , Modulo JAR dentro de un archivo EAR, Archivo JAR de aplicación cliente, Directorio WEB-INF/classes de un archivo WAR

- El descriptor persistence.xml define:
  - El nombre de la unidad de persistencia
  - El tipo de gestión de transacciones que se soporta
    - JTA cuando estamos en Java EE (gestionadas por el contenedor)
    - RESOURCE_LOCAL, cuando estamos en Java SE (hay que gestionarlas a mano)
  - Propiedades de conexión con la base de datos
    - Usando JTA, basta con indicar el nombre de la “data-source”, que debe estar definida en el correspondiente servidor
  - Clases entidad que forman la unidad. Dos opciones de declaración:
    - Basado en descriptor: Las clases entidad se indican explícitamente en el descriptor
    - Basado en anotaciones: Todas las clases incluidas en el .jar y anotadas con @Entity

# Entity Manager: Ejemplo de descriptor persistence.xml




```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
   <persistence-unit name="VehiculosPU" transaction-type="JTA">
       <jta-data-source>jdbc:/itv</jta-data-source>

       <!- Sólo si las clases no están anotadas -->
       <class>itv.entidades.Propietario</class>
       <class>itv.entidades.Vehiculo</class>

       <!- Asignación de propiedades propias del proveedor de persistencia -->
       <properties>
           <!-- Standard and Provider-specific config may go here -->
           <property name="myprovider.property.name" value="someValue"/>
       </properties>
   </persistence-unit>
</persistence>
```

# Entity Manager: Acceso desde Java EE

- En aplicaciones Java EE, el acceso al EntityManager se consigue a través de inyección de dependencias
  - Indicando el contexto de persistencia que se quiere manejar (identificado por el nombre de su correspondiente unidad de persistencia)
  - Las transacciones son gestionadas por el contenedor
  - El fichero persistence.xml se debe incluir en el despliegue

**Inyección del EntityManager** →

```
@Stateless
public class VehiculosDAOBean implements VehiculosDAOLocal, VehiculosDAORemote {

   @PersistenceContext(unitName="VehiculosPU");
   private EntityManager em;

   public boolean creaVehiculo(Vehiculo v) {
      // Hacemos persistente el vehículo(comprobando que no existía previamente)
      if (em.find(Vehiculo.class, v.getMatricula()) == null)  {
            em.persist(a);
            return true;
      } else {
            return false;
      }
   }
}
```

# Entity Manager: Acceso desde Java SE

- En aplicaciones Java SE, se puede obtener a través de una clase Factory
  - Hay que controlar las transacciones de manera explícita y la propia vida del Entity Manager
  - El fichero persistence.xml debe estar accesible en el classpath

```
public static void main(String[] args) {
   // Creación de la entidad
   Vehiculo v = new Vehiculo(“7777XGH“, “Toyota”);

   // Obtención del entity Manager
   EntityManagerFactory emf = Persistence.createEntityManagerFactory(“VehiculosPU");
   EntityManager em = emf.createEntityManager();

   // La entidad se hace persistente (dentro de transacción)
   EntityTransaction tx = em.getTransaction();
   tx.begin();
   em.persist(book);
   tx.commit();

   // Liberación de recursos
   em.close();
   emf.close();
}
```

# Diseño y desarrollo de entidades

- Existen dos estrategias para la implementación de entidades en una aplicación Java EE
  - Partir del diseño de objetos, anotarlo debidamente y generar automáticamente la base de datos
  - Partir de la BBDD y generar automáticamente las clases entidad

- Los proveedores de persistencia avanzados suelen ofrecer ambas capacidades

# JPQL: Java Persistence Query Language

- **JPQL** es un lenguaje definido en JPA para realizar consultas en la BBDD
  - Independiente del servidor de BBDD subyacente
  - Sintaxis similar a SQL pero orientada a objetos (centrada en entidades)
    - La clausula FROM se refiere a entidades y los parámetros de búsqueda a atributos de las entidades
  - Soporta sentencias SELECT, UPDATE y REMOVE

- Ejemplos de sintaxis
  - Búsqueda de todas las instancias de una entidad
    - `SELECT v FROM Vehiculo v`
    - `SELECT p FROM PROPIETARIO p ORDER BY p.edad DESC`
  - Búsqueda de todas las instancias de una entidad restringiendo algún atributo
    - `SELECT v FROM VEHICULO v WHERE v.modelo = 'Toyota Auris'`
    - `SELECT p FROM PROPIETARIO p WHERE p.edad > 35`
    - `SELECT p FROM PROPIETARIO p WHERE p.direccion.localidad = 'Santander'`
    - `SELECT p FROM PROPIETARIO p WHERE p.direccion.localidad = 'Santander' AND p.edad > 35`
  - Búsqueda del atributo(s) de las instancias de una determinada entidad restringiendo un atributo
    - `SELECT v.matricula FROM VEHICULO v WHERE v.modelo = 'Toyota Auris'`

# JPQL: Consultas a través de Clase Query

- Las consultas JPQL se realizan a través de objetos de la clase **Query**

- Existen tres tipos de Query:
  - Dynamic Query: Consulta JPQL que se genera en tiempo de ejecución
  - Named Query: Consulta JPQL estática y no modificable
  - Native Query: Consultas nativas SQL

- La clase EntityManager proporciona métodos para la creación de objetos Query

```
public interface EntityManager {

     // Crea una instancia de una consulta JPQL dinámica
     public Query createQuery(String jpqlString);

     // Crea una instancia de una consulta JPQL con nombre
     public Query createNamedQuery(String jpqlString);

     // Crea una instancia de una consulta SQL nativa
     public Query createNativeQuery(String sqlString);

     ...
}
```

# JPQL: Ejecución de consultas dinámicas

- La interfaz **Query** proporciona métodos para la ejecución de consultas

```
public interface Query {

    // Ejecuta una sentencia SELECT y retorna la lista de objetos resultante
    // Retorna null si no hay resultados
    public List<Object> getResultList();

    // Ejecuta una sentencia SELECT y retorna el objeto resultado
    // Lanza NoResultException si no hay resultado
    // Lanza NonUniqueResultException si hay más de un resultado
    public Object getSingleResult();

    // Ejecuta una sentencia UPDATE/REMOVE y retorna el número de entidades afectadas
    public int executeUpdate();
    ...
}
```

```
public class VehiculosDAO () {

   @PersistenceContext("VehiculosPU")
   private EntityManager em;

   public List<Vehiculo> vehiculos() {
          Query q = em.createQuery("SELECT v FROM Vehiculo v");
          List<Vehiculo> vehiculos = q.getResultList();
          if (vehiculos != null)
                return vehiculos;
          return null;
   }
}
```

# JPQL: Consultas parametrizadas

- Sintaxis JPQL
  - Parámetros posicionales

```
SELECT v FROM VEHICULO v WHERE v.modelo = ?1
```

  - Parámetros con nombre

```
SELECT v FROM VEHICULO v WHERE v.modelo = :modelo
```

- Ejecución de consultas (métodos setParameter de la interfaz Query)

```
// Asocia el valor o al parámetro de índice position
public void setParameter(int position, Object o);
// Asocia el valor o al parámetro de nombre name
public void setParameter(String name, Object o);
```

```
public class VehiculosDAO () {

   @PersistenceContext("VehiculosPU")
   EntityManager em;

   public Vehiculo vehiculoPorMatricula(String matricula) {
          Query q = em.createQuery("SELECT v FROM Vehiculo v WHERE v.matricula = :mat");
          q.setParameter("mat", matricula);
          List<Vehiculo> vehiculos = q.getResultList();
          if (vehiculos != null)
                    return vehiculos;
          return null;
   }
}
```

# JPQL: Consultas estáticas (Named Queries)

- Consultas estáticas y no modificables
- Más eficientes que las consultas dinámicas
- Definición: A través de la anotación `@NamedQuery` en las entidades
  - Los nombres deben ser únicos a nivel de unidad de persistencia
- Ejecución: Método `createNamedQuery()`

```
@Entity
@NamedQueries( {
    @NamedQuery(name="vehiculos", query="SELECT v FROM Vehiculo v"),
    @NamedQuery(name="vehiculosPorMatricula" query="SELECT v FROM VEHICULO v WHERE v.matricula = :mat')
})

public class Vehiculo {
   ...
}
```

```
public class VehiculosDAO {

   @PersistenceContext("VehiculosPU")
   private EntityManager em;

   public List<Vehiculo> vehiculos() {
       Query q = em.createNamedQuery("vehiculosPorMatricula");
       q.setParameter("mat", "1111 AAA");
       List<Vehiculo> vehiculos = q.getResultList();
       if (vehiculos != null)
           return vehiculos;
       return null;
   }
}
```