



# **Persistencia y gestión de datos con JPA**

## **Sesión 3: Mapeo entidad-relación (relaciones)**



# Índice

- Conceptos previos sobre relaciones
- Definición de los distintos tipos de relaciones
- Carga perezosa



# Relaciones entre entidades

- En las relaciones se asocia una instancia de una entidad A con una o muchas instancias de otra entidad B
- En JPA se construyen las relaciones de una forma natural: definiendo en la instancia A el atributo de la relación con el tipo de la otra entidad B o como una colección de instancias de B
- Habitualmente las relaciones que se definen en JPA son bidireccionales, pudiendo obtener la entidad A a partir de la B y la B a partir de la A

```
@Entity
public class Empleado {
    // ...
    @OneToOne
    private despacho Despacho;
    // ...

    public void setDespacho(Despacho despacho) {
        this.despacho = despacho;
    }
}
```

```
@Entity
public class Despacho {
    @Id
    String idDespacho;

    // ...
}
```



# Cardinalidad

- La cardinalidad de una relación define si asociada a una instancia A hay una o muchas instancias B
- La relación inversa define también la cardinalidad: ¿la misma instancia de B puede ser el destino de más de una instancia A?
- Dependiendo de las respuestas tenemos tres posibles tipos de relaciones bidireccionales (las relaciones inversas entre paréntesis)
  - one-to-one (inversa: one-to-one)
  - one-to-many (inversa: many-to-one)
  - many-to-many (inversa: many-to-many)



# Cardinalidad y direccionalidad en JPA

- En JPA definimos la cardinalidad de una relación anotando los atributos que intervienen en la misma
- Posibles anotaciones
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany
- La direccionalidad depende de si se define o no el atributo relacionado con A en la entidad B
- En el caso en que la relación fuera bidireccional, la relación inversa debe tener también la asociación inversa



# Mapeo de una relación

- Existen básicamente dos formas de mapear una relación en SQL
  - utilizando claves ajenas en una de las tablas
  - utilizando una tabla adicional (join) que implementa la asociación
- Las tablas con claves ajenas pueden implementar las relaciones one-to-one y one-to-many (y la many-to-one bidireccional)
- La tabla join se usa para implementar las asociaciones many-to-many

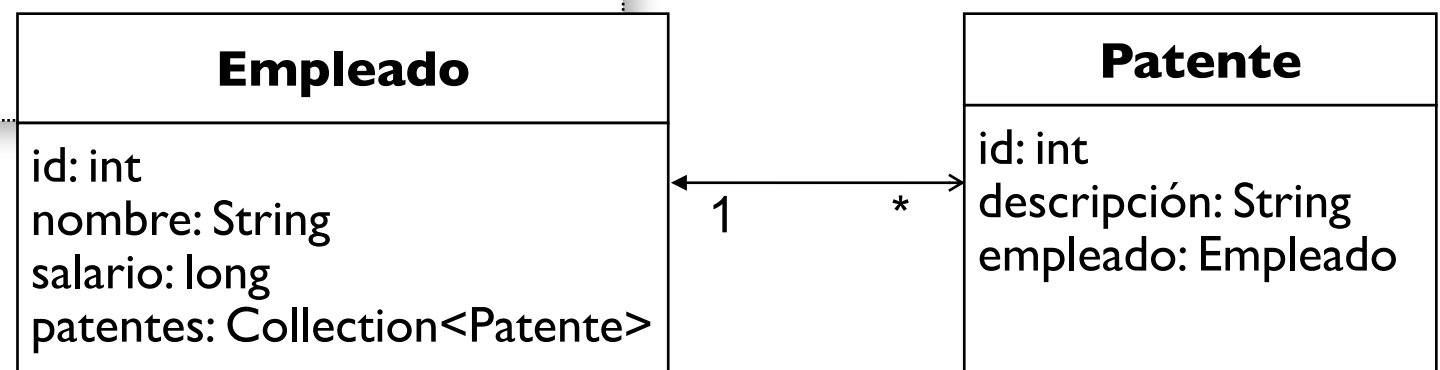


# Un ejemplo

- Una relación bidireccional one-to-many

```
@Entity
public class Empleado {
    @Id String nombre;
    @OneToMany(mappedBy="empleado")
    private Set<Patentes> patentes = new HashSet();
    ...
}

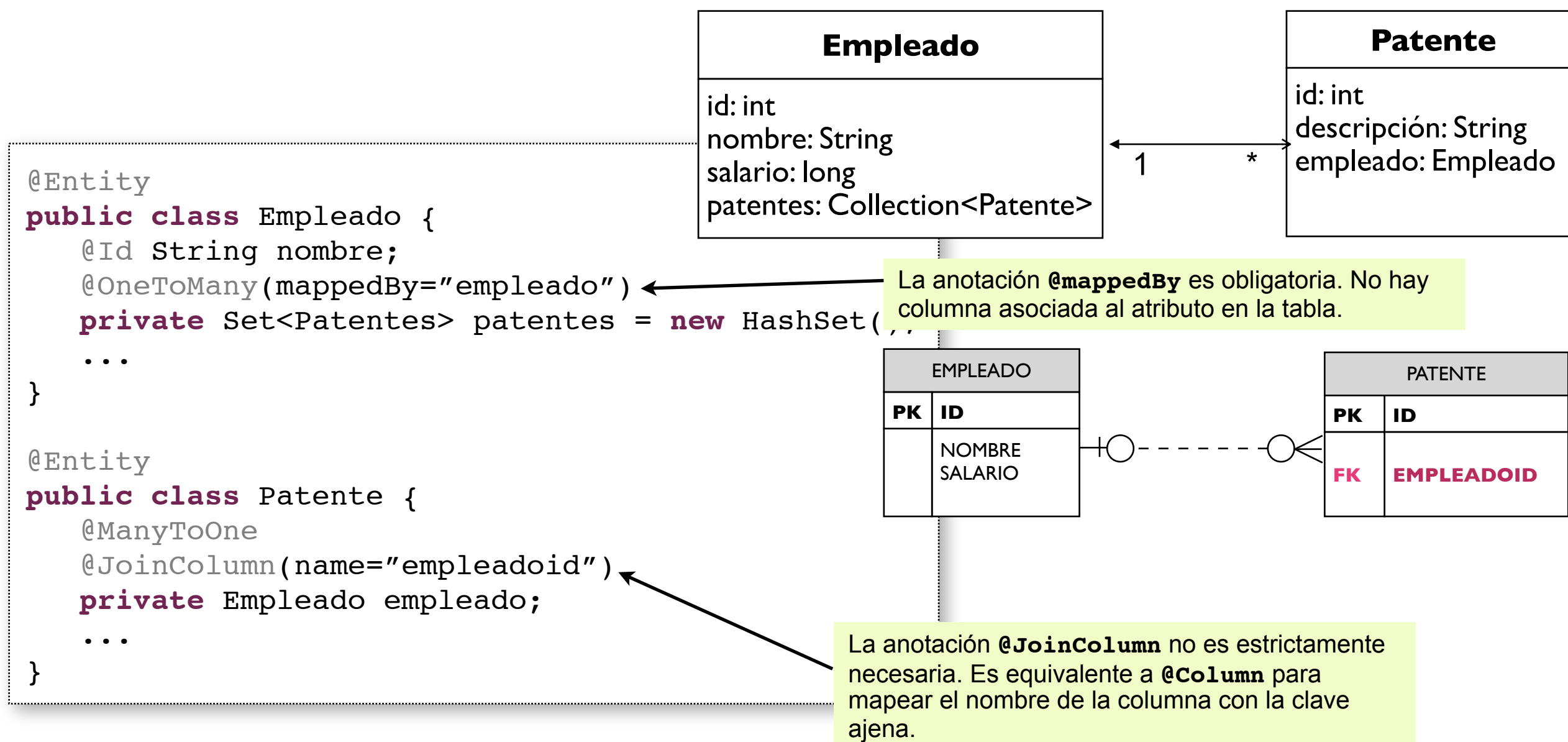
@Entity
public class Patente {
    @ManyToOne
    @JoinColumn(name="empleadoid")
    private Empleado empleado;
    ...
}
```





# Mapeo con clave ajena

- En JPA se indica la tabla que contiene la clave ajena (la entidad propietaria de la relación) con el elemento `mappedBy` en el otro lado de la relación







# One-to-one unidireccional

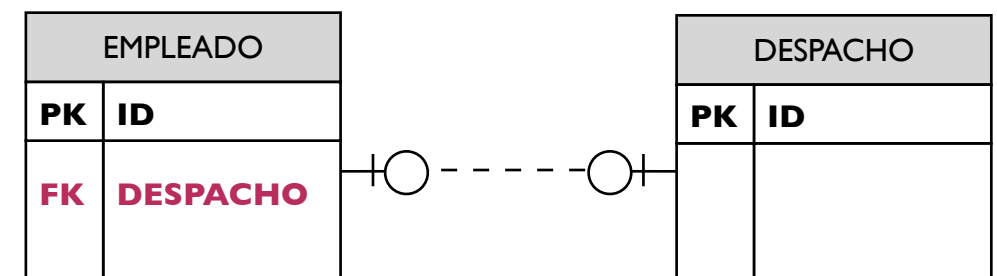
```
@Entity
public class Empleado {
    // ...
    @OneToOne
    private despacho Despacho;
    // ...

    public void setDespacho(Despacho despacho) {
        this.despacho = despacho;
    }
}

@Entity
public class Despacho {
    @Id
    String idDespacho;

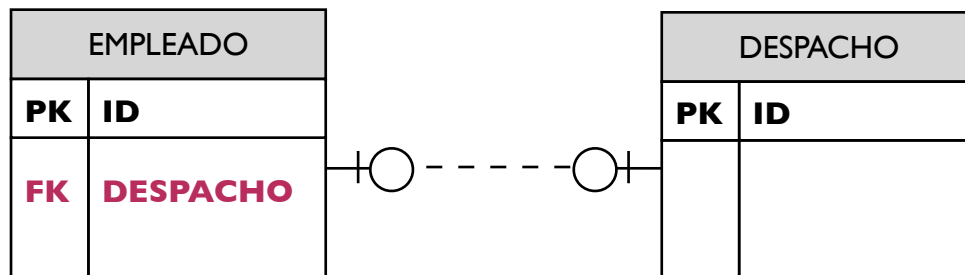
    // ...
}
```

No se usa la anotación **@JoinColumn** porque el nombre de la columna coincide con el del atributo (despacho).





# One-to-one bidireccional



```
@Empleado
public class Empleado {
    // ...
    @OneToOne
    private despacho Despacho;
    // ...

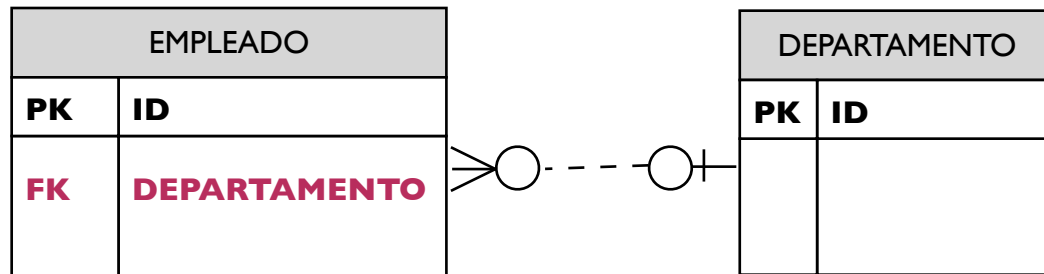
    public void setDespacho(Despacho despacho) {
        this.despacho = despacho;
    }
}

@Entity
public class Despacho {
    // ...
    @OneToOne(mappedBy="despacho")
    private empleado Empleado;
    // ...

    public Empleado getEmpleado() {
        return this.empleado;
    }
}
```



# One-to-many (Many-to-one) bidireccional



Clave ajena

```
@Entity
public class Empleado {
    // ...
    @ManyToOne
    @JoinColumn(name="departamento")
    private departamento Departamento;
    // ...
}

@Entity
public class Departamento {
    // ...
    @OneToMany(mappedBy="departamento")
    private Set<Empleado> empleados = new HashSet();
    // ...
}
```



# Actualización de las relaciones

- A efectos de la BD y de las posteriores queries, bastaría con actualizar la entidad propietaria de la relación, la que contiene la clave ajena (en el caso del ejemplo, el empleado)
- Problema en memoria: si el entity manager ya está gestionando la colección inversa (el conjunto de empleados del departamento) no va a volver a consultarla de la BD (a no ser que hagamos un refresh de la entidad)
- Solución: es conveniente siempre actualizar en memoria la colección

```
emp.setDepartamento(depto);  
depto.getEmpleados().add(emp);
```



# Many-to-many bidireccional (anotaciones)

```
@Entity
public class Empleado {
    @Id String nombre;
    @ManyToMany
    private Set<Proyecto> proyectos = new HashSet();

    public Set<Proyecto> getProyectos() {
        return this.proyectos;
    }

    public void setProyectos(Set<Proyecto> proyectos) {
        this.proyectos = proyectos;
    }

    // ...
}
```

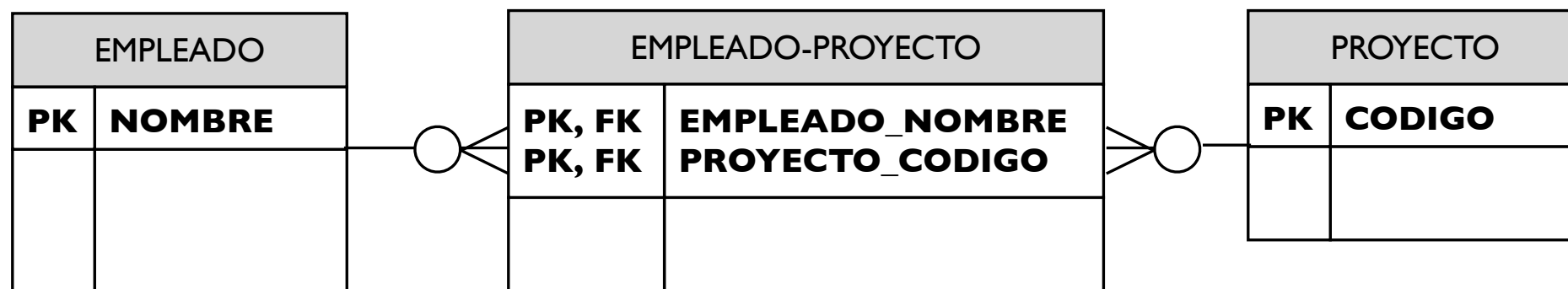
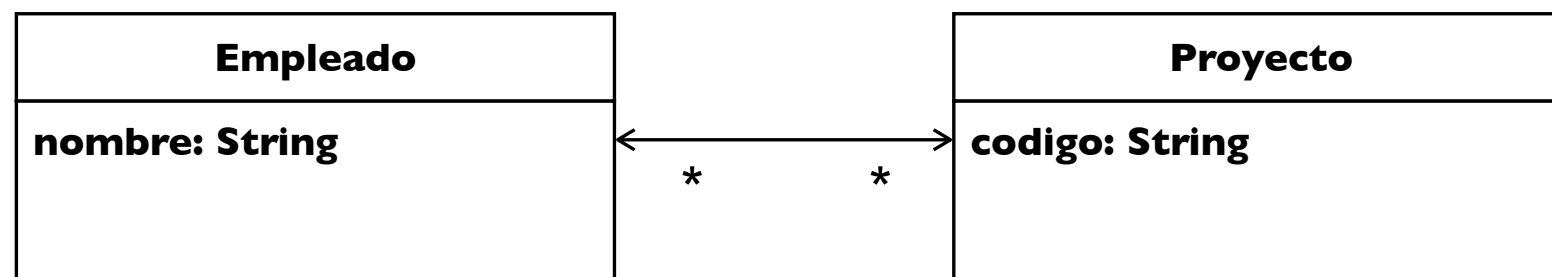
```
@Entity
public class Proyecto {
    @Id String codigo;
    @ManyToMany(mappedBy="proyectos");
    private Set<Empleado> empleados = new HashSet();

    public Set<Empleado> getEmpleados() {
        return empleados;
    }

    // ...
}
```



# Many-to-many bidireccional (mapeo)





# Anotación JoinTable

- Si queremos especificar el nombre y características de la tabla join, podemos utilizar la anotación @JoinTable:

```
@Entity
public class Empleado {
    @Id String nombre;
    @ManyToMany
    @ManyToMany
    @JoinTable(
        name = "EMPLEADO_PROYECTO",
        joinColumns = {@JoinColumn(name = "EMPLEADO_NOMBRE")},
        inverseJoinColumns = {@JoinColumn(name = "PROYECTO_CODIGO")}
    )
    private Set<Proyecto> proyectos = new HashSet<Proyecto>();
}
```



# Actualización de relaciones (1)

- Siempre se sincroniza con la base de datos la entidad que contiene la clave ajena (se le llama entidad propietaria de la relación)
- La otra entidad conviene actualizarla también para mantener la relación consistente en memoria
- Añadiendo un empleado a un proyecto:

```
Proyecto proyecto = em.find(Proyecto.class, "P04");  
Set<Proyectos> proyectos = empleado.getProyectos();  
proyectos.add(proyecto);
```

- Cambiando un empleado de proyectos;

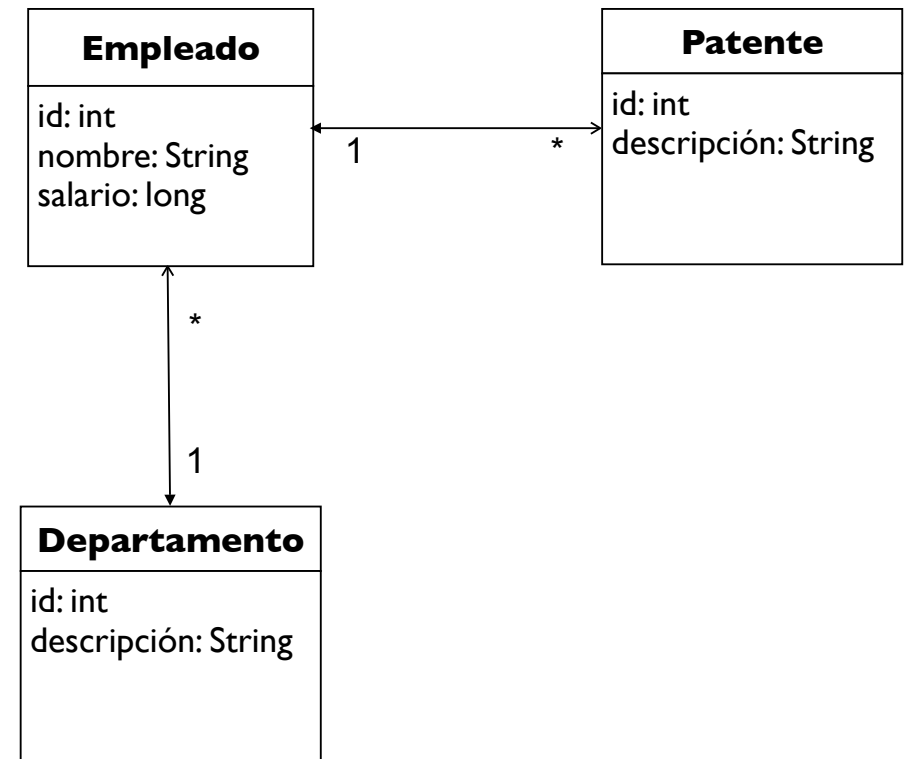
```
Empleado empleado = em.find(Empleado.class, "Miguel Garcia");  
Proyecto proyectoBaja = em.find(Proyecto.class, "P04");  
Proyecto proyectoAlta = em.find(Proyecto.class, "P01");  
empleado.getProyectos().remove(proyectoBaja);  
empleado.getProyectos().add(proyectoAlta);
```





# Carga perezosa

- Al recuperar un departamento sería muy costoso recuperar todos sus empleados
- En las relaciones “a muchos” JPA utiliza la carga perezosa por defecto
- Es posible desactivar este comportamiento con la opción `fetch=FetchType.EAGER` en el tipo de relación
- Un empleado no tiene muchas patentes

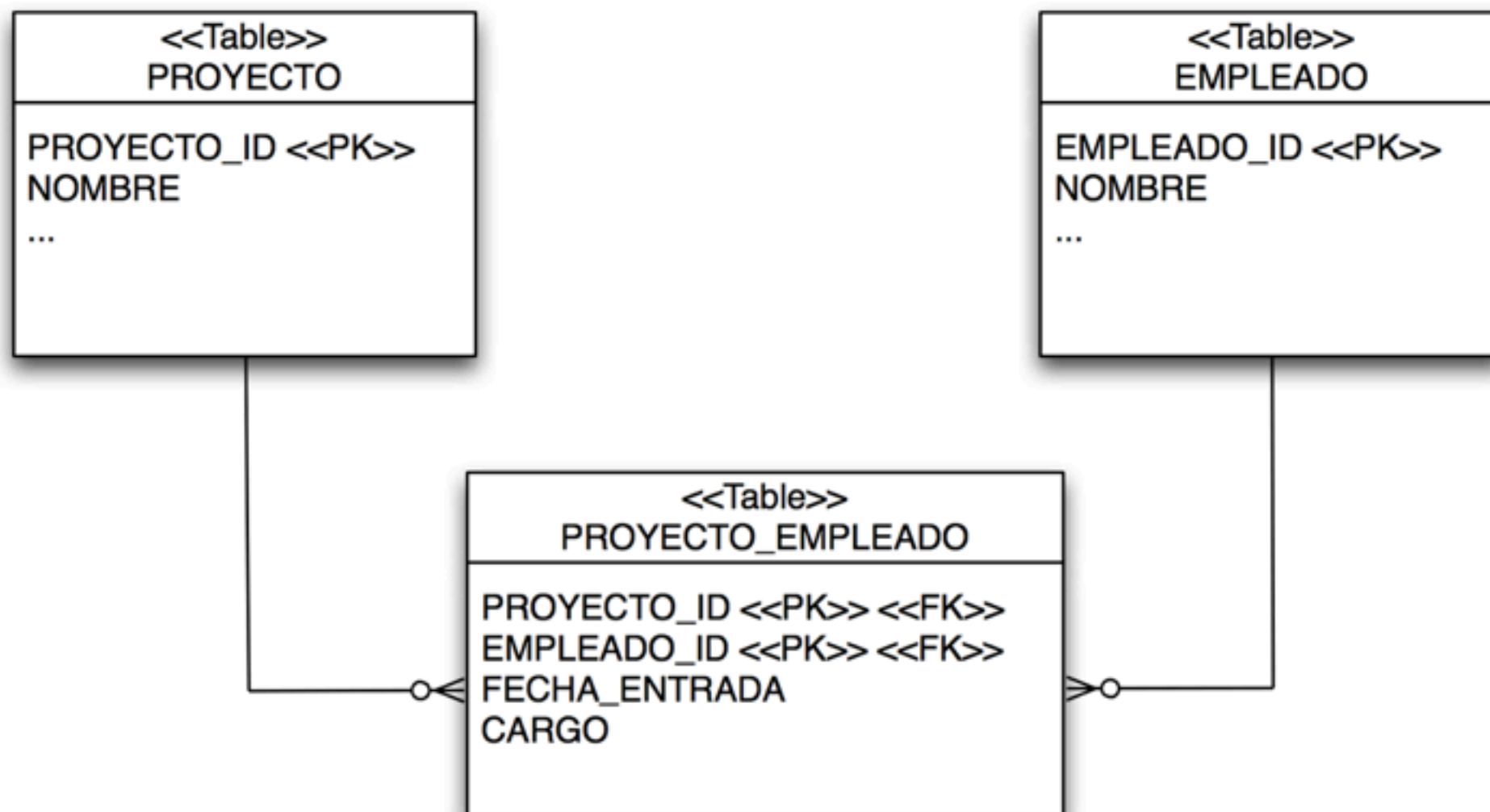


```
@Entity
public class Empleado {
    @Id String nombre;
    @OneToMany(fetch=FetchType.EAGER)
    private Set<Patentes> patentes = new HashSet();
    // ...
}
```



# Columnas adicionales en la tabla join

- Es muy usual en esquemas heredados usar columnas adicionales en las tablas join





# Mapeado

- Hay que crear una entidad adicional (ProyectoEmpleado) con una clave primaria compuesta y una relación uno-a-muchos con las otras entidades (Proyecto y Empleado)
- En esta entidad adicional se mapean los atributos de las claves primarias y los de las relaciones en las mismas columnas, con los atributos `insertable = false` y `updatable = false`
- Consultar en los apuntes un ejemplo completo



# ¿Preguntas?