Frameworks de persistencia - JPA

Sesión 4 - Mapeado de relaciones

Índice

Relaciones entre entidades

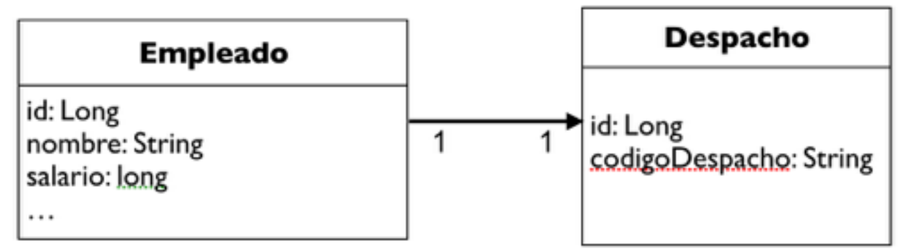
- En las relaciones se asocia una instancia de una entidad A con una o muchas instancias de otra entidad B
- En JPA se construyen las relaciones de una forma natural: definiendo en la instancia A el atributo de la relación con el tipo de la otra entidad B o como una colección de instancias de B
- Habitualmente 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

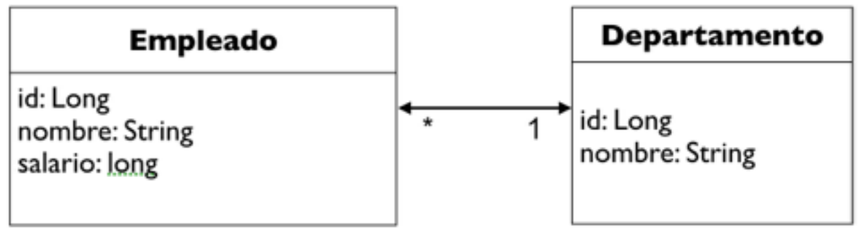
```
@Entity
public class Empleado {
    ...
    @OneToOne
    private despacho Despacho;
    ...

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

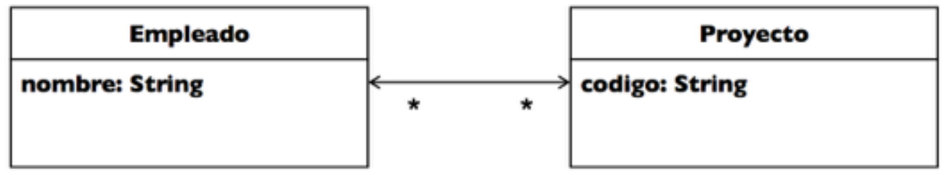
```
@Entity
public class Despacho {
    ...
    Empleado empleado;
    ...
}
```

Cardinalidad y direccionalidad

- 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 desde el punto de vista de la entidad de la izquierda. Desde el punto de vista de la entidadde la derecha tenemos la relación inversa.
  - uno-a-uno (inversa: uno-a-uno): **@OneToOne**
  - uno-a-muchos (inversa: muchos-a-uno): **@OneToMany**, (**@ManyToOne**)
  - muchos-a-muchos (inversa: muchos-a-muchos): **@ManyToMany**



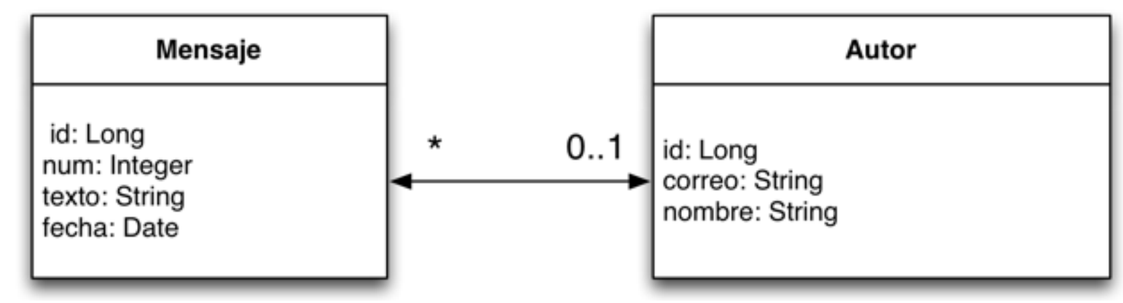


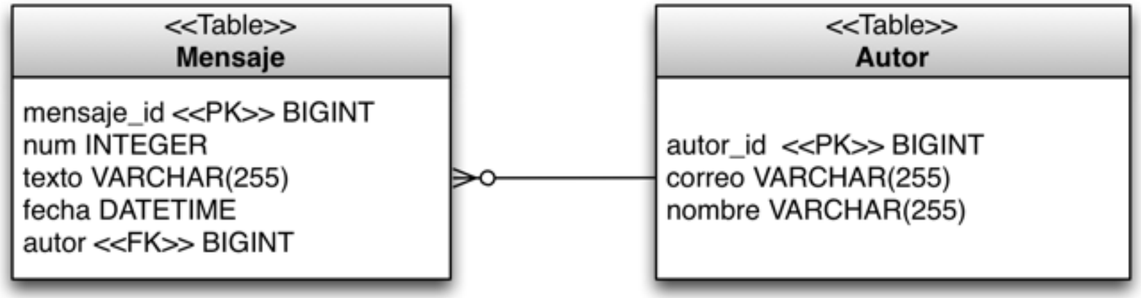


- La direccionalidad indica si desde una entidad podemos obtener (*gettter*) la otra.

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
- La entidad propietaria de la relación es la que se mapea con la tabla que tiene la clave ajena





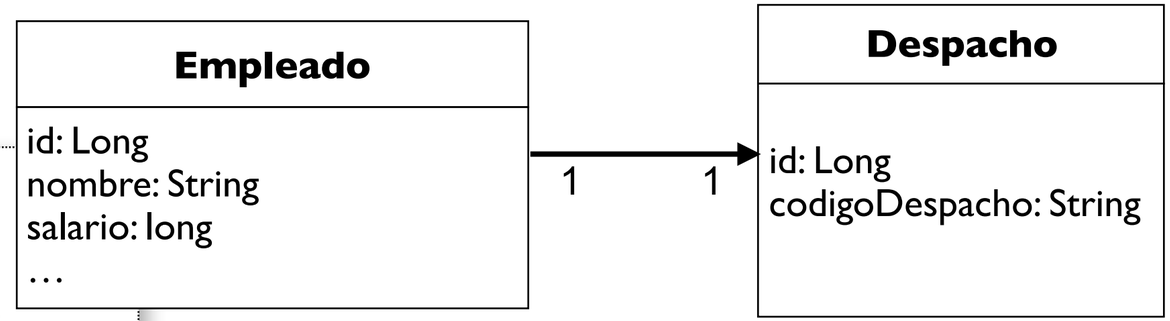
Actualización de una relación

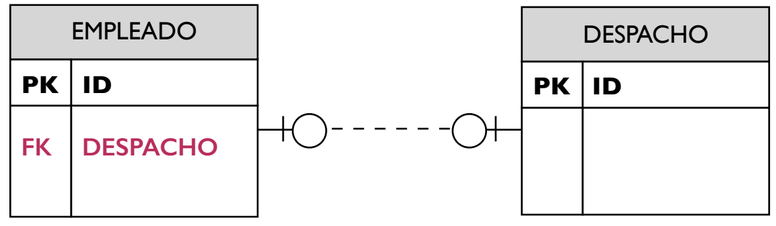
- Usar los métodos setters para reasignar las instancias que aparecen en la relación.
- Actualización de las tablas mapeadas
  - Es necesario actualizar la entidad propietaria de la relación
  - Deben respetarse las restricciones definidas en las columnas (unique, not null, …)
- Actualización del grafo de objetos en memoria
  - JPA no se encarga de actualizar la relación inversa (excepto en las relaciones muchos-a-muchos)
  - Es necesario que actualicemos la relación inversa para mantener consistente el grafo de objetos en memoria
- Es recomendable implementar métodos auxiliares en las entidades

```
// añadimos un mensaje a un autor
// se actualiza el campo autor del mensaje
// imprescindible para actualizar la BD
mensaje.setAutor(autor);
// actualizamos la relación inversa en memoria
autor.getMensajes().add(mensaje);
```

Uno-a-uno unidireccional

```java
public class Empleado {
    ...
    @OneToOne
    @JoinColumn(name = "despacho_id", unique = true)
    private Despacho despacho;
    ...

    public Despacho getDespacho() {
        return despacho;
    }

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


@Entity
public class Despacho {
   // ...
}
```



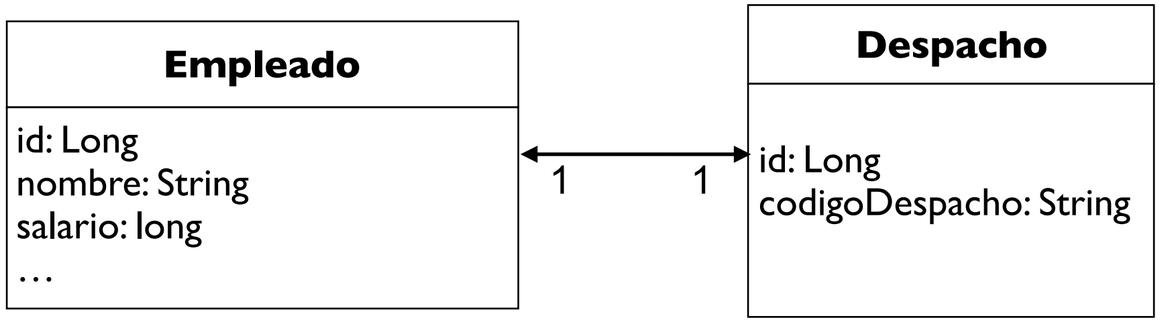


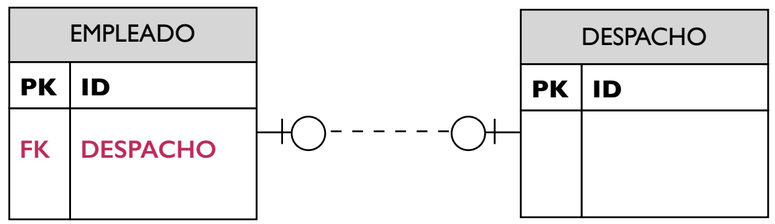
Actualización de la relación

```java
em.getTrasaction().begin();
Empleado empleado = em.find(Empleado.class, 1L);
Despacho despacho = em.find(Despacho.class, 3L);
empleado.setDespacho(despacho);
em.getTransaction().commit();
```

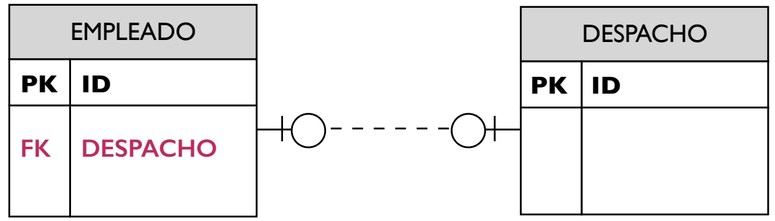
Uno-a-uno bidireccional

```
@Empleado
public class Empleado {
    ...
    @OneToOne
    @JoinColumn(name = "despacho_id", unique = true)
    private despacho Despacho;
    ...
}

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





Métodos auxiliares



Empleado

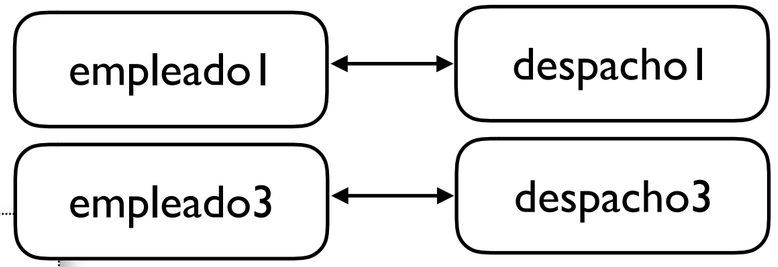
```
@Entity
public class Empleado {
    ....
    public Despacho getDespacho() {
        return despacho;
    }

    public void setDespacho(Despacho despacho) {
        this.despacho = despacho;
        if (despacho.getEmpleado() != this) {
            despacho.setEmpleado(this);
        }
    }

    public void quitaDespacho() {
        if (this.getDespacho() != null) {
            this.getDespacho().quitaEmpleado();
            this.despacho = null;
        }
    }
    ...
}
```

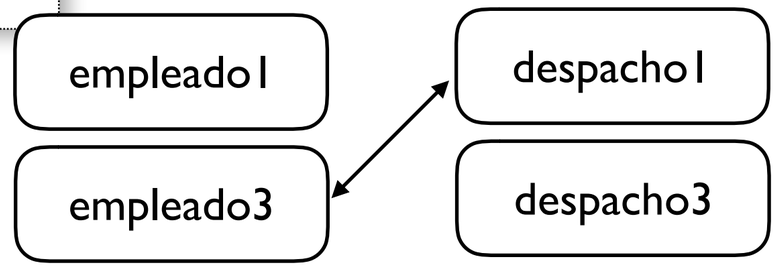
Despacho

```
@Entity
public class Despacho {
    ...
    public void setEmpleado(Empleado empleado) {
        this.empleado = empleado;
        empleado.setDespacho(this);
    }

    public void quitaEmpleado() {
        if (this.empleado != null) {
            this.empleado.quitaDepartamento();
        }
        this.empleado = null;
    }
    ...
}
```

Ejemplo actualización relaciones

```
// El empleado1 está en el despacho1. Ponemos en el despacho
// del empleado 1 al empleado 3 (dejando sin despacho al empleado 1)

em.getTransaction().begin();
Empleado empleado1 = em.find(Empleado.class, 1L);
Despacho despacho1 = empleado1.getDespacho();
Empleado empleado3 = em.find(Empleado.class, 3L);
empleado1.quitaDespacho();
empleado3.setDespacho(despacho1);
em.getTransaction().commit();
```



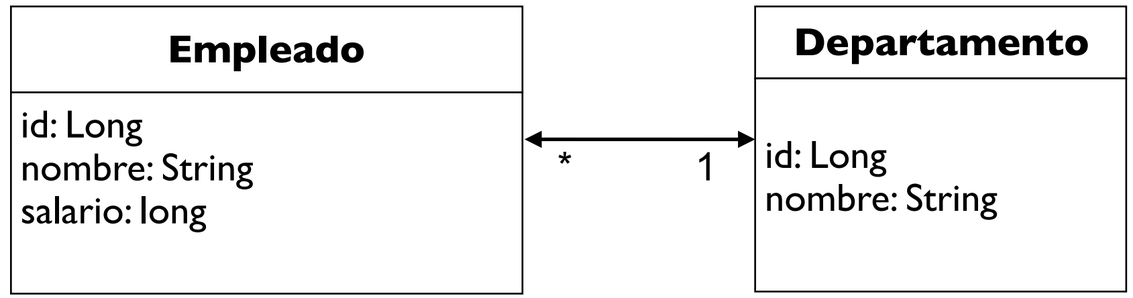


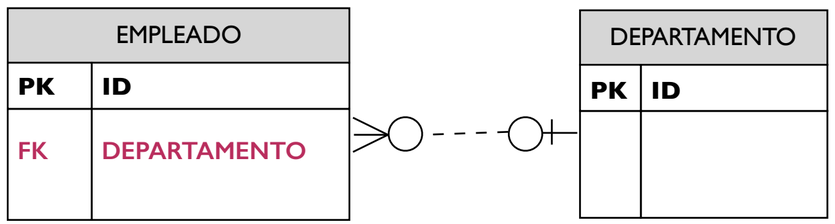
Relación bidireccional muchos-a-uno (y uno-a-muchos)

La anotación **`@JoinColumn`** no es estrictamente necesaria. Es equivalente a **`@Column`** para mapear el nombre de la columna con la clave ajena.

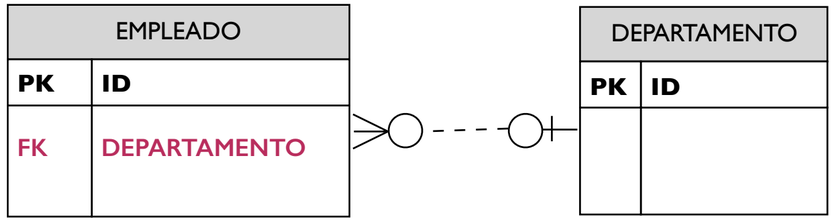
```
@Entity
public class Empleado {
    ...
    @ManyToOne
    @JoinColumn(name="departamento_id")
    private departamento Departamento;
    ...
}

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

La anotación **`@mappedBy`** es obligatoria. Indica que no hay columna asociada al atributo en la tabla.





Métodos auxiliares



Empleado

```
@Entity
public class Empleado {
    ...

    public Departamento getDepartamento() {
        return departamento;
    }

    public void setDepartamento(Departamento departamento) {
        this.departamento = departamento;
        departamento.getEmpleados().add(this);
    }

    public void quitaDepartamento() {
        Departamento departamento = this.getDepartamento();
        this.departamento = null;
        departamento.getEmpleados().remove(this);
    }
    ...
}
```

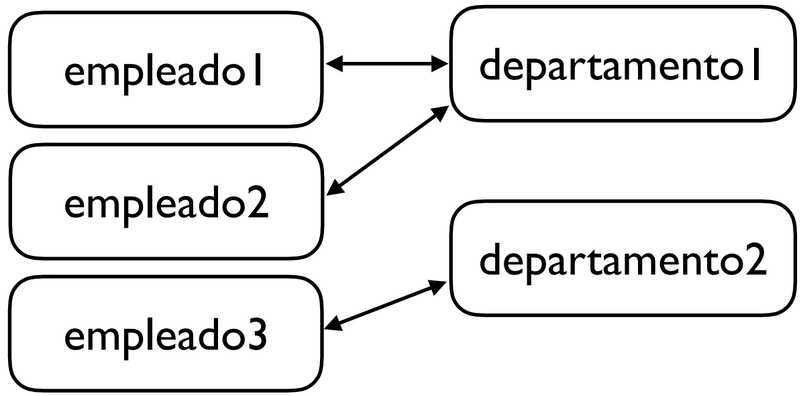
Departamento

```
@Entity
public class Departamento {
    ...

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

    public void añadeEmpleado(Empleado empleado) {
        this.getEmpleados().add(empleado);
        empleado.setDepartamento(this);
    }

    public void quitaEmpleado(Empleado empleado) {
        this.getEmpleados().remove(empleado);
        empleado.quitaDepartamento();
    }
    ...
}
```

Ejemplo actualización relaciones

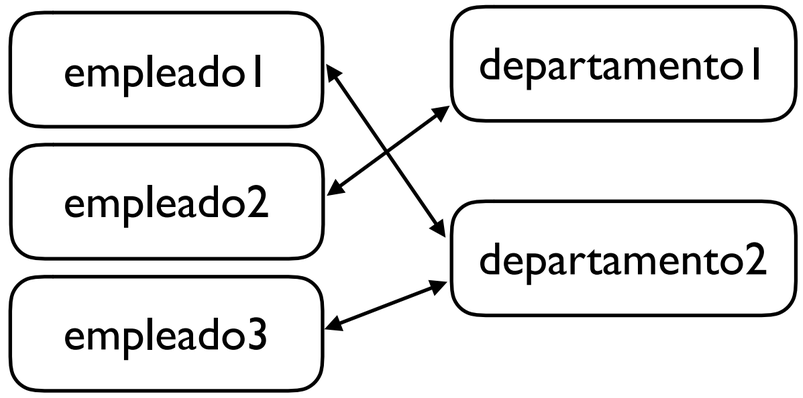
Usando los métodos de Empleado

```
// El empleado1 pertenece al departamento1.
// Lo cambiamos al departamento2.

Departamento depto1 = em.find(Departamento.class, 1L);
Departamento depto2 = em.find(Departamento.class, 2L);
Empleado empleado1 = em.find(Empleado.class, 1L);
empleado1.quitaDepartamento();
empleado1.setDepartamento(depto2);
```



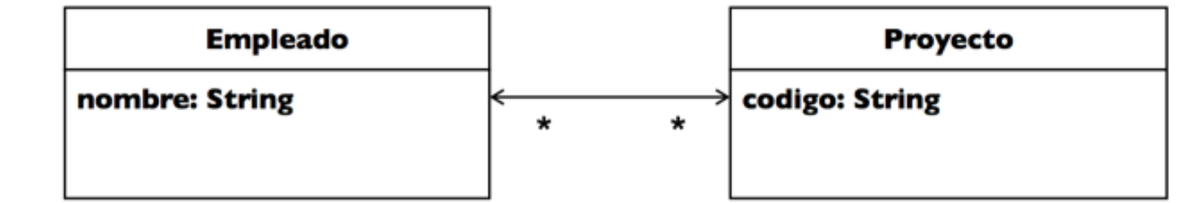
Usando los métodos de Departamento

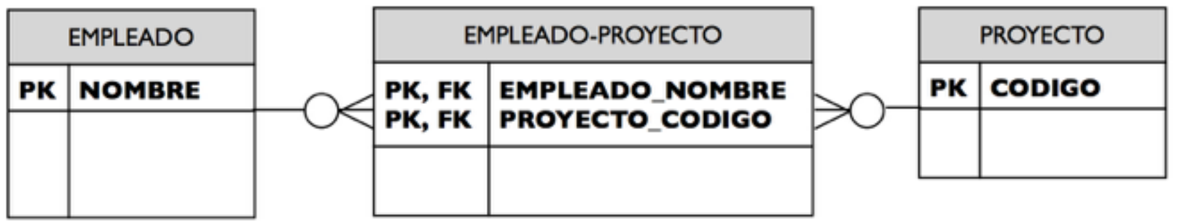
```
// El empleado1 pertenece al departamento1.
// Lo cambiamos al departamento2.

Departamento depto1 = em.find(Departamento.class, 1L);
Departamento depto2 = em.find(Departamento.class, 2L);
Empleado empleado1 = em.find(Empleado.class, 1L);
depto1.quitaEmpleado(empleado1);
depto2.añadeEmpleado(empleado1);
```



Muchos-a-muchos bidireccional

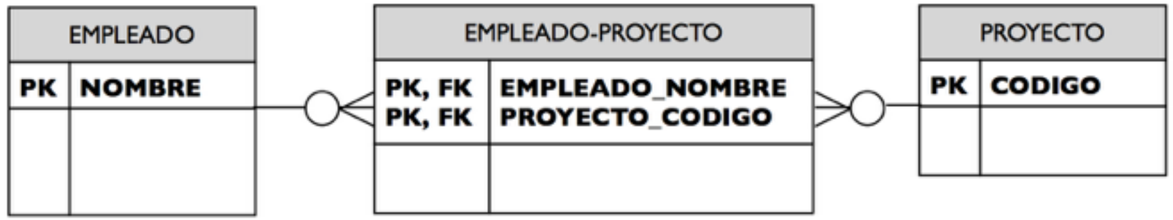
```java
@Entity
public class Empleado {
    ...
    @ManyToMany
    private Set<Projecto> proyectos = new HashSet();
    ...

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

    ...
}
```






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

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

Métodos auxiliares



Empleado

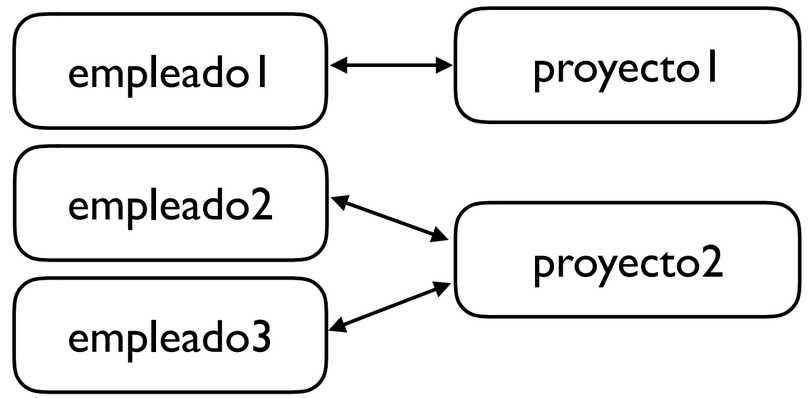
```
@Entity
public class Empleado {
    ...
    public Set<Proyecto> getProyectos() {
        return proyectos;
    }

    public void añadeProyecto(Proyecto proyecto) {
        this.getProyectos().add(proyecto);
        proyecto.getEmpleados().add(this);
    }

    public void quitaProyecto(Proyecto proyecto) {
        this.getProyectos().remove(proyecto);
        proyecto.getEmpleados().remove(this);
    }

    ...
}
```

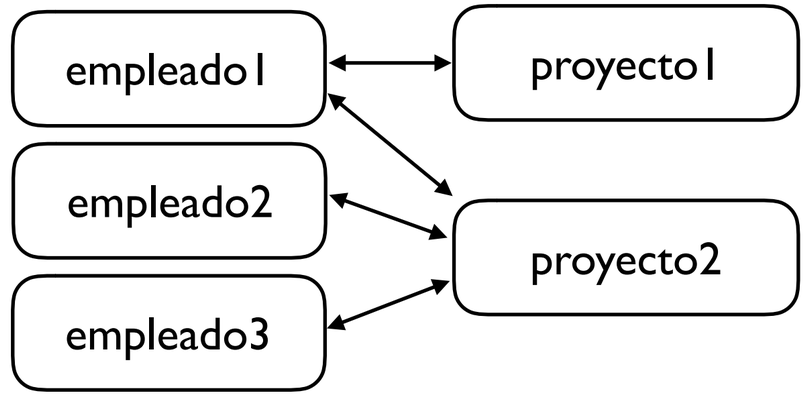
Proyecto

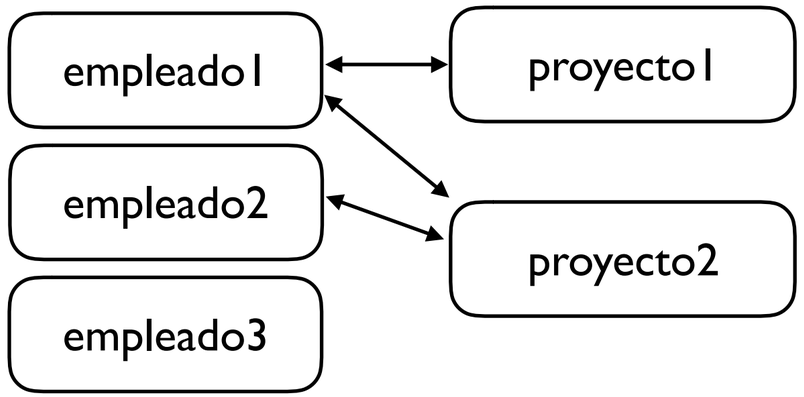
```
@Entity
public class Proyecto {
    ...
    public Set<Empleado> getEmpleados() {
        return empleados;
    }

    public void añadeEmpleado(Empleado empleado) {
        empleado.getProyectos().add(this);
        this.getEmpleados().add(empleado);
    }

    public void quitaEmpleado(Empleado empleado) {
        empleado.getProyectos().remove(this);
        this.getEmpleados().remove(empleado);
    }
    ...
}
```

Ejemplo actualización relaciones

```
// Añadimos el empleado 1 al proyecto 2

Proyecto proyecto2 = em.find(Proyecto.class, 2L);
Empleado empleado1 = em.find(Empleado.class, 1L);
empleado1.añadeProyecto(proyecto2);

// Y eliminamos el empleado 3 del proyecto 2

Empleado empleado3 = em.find(Empleado.class, 3L);
empleado3.quitaProyecto(proyecto2);
```







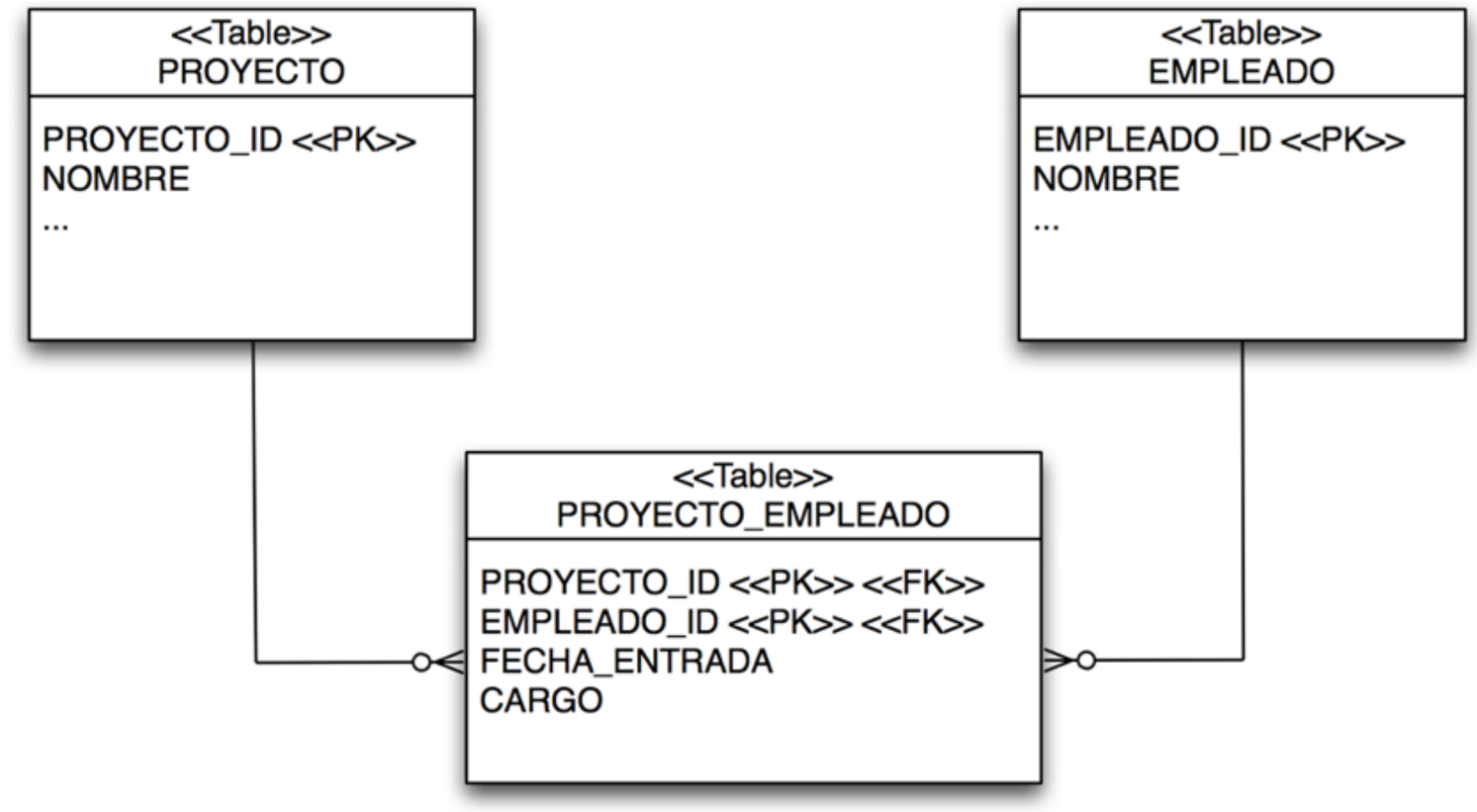
Anotación JoinTable

- 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<Projecto> proyectos = new HashSet<Projecto>();
```

Columnas adicionales en la tabla join

- Es muy usual en esquemas heredados usar columnas adicionales en las tablas join
- Se puede definir en JPA definiendo explícitamente la tabla join con una clave primaria compuesta por las dos claves ajenas
- 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`



Código (1)

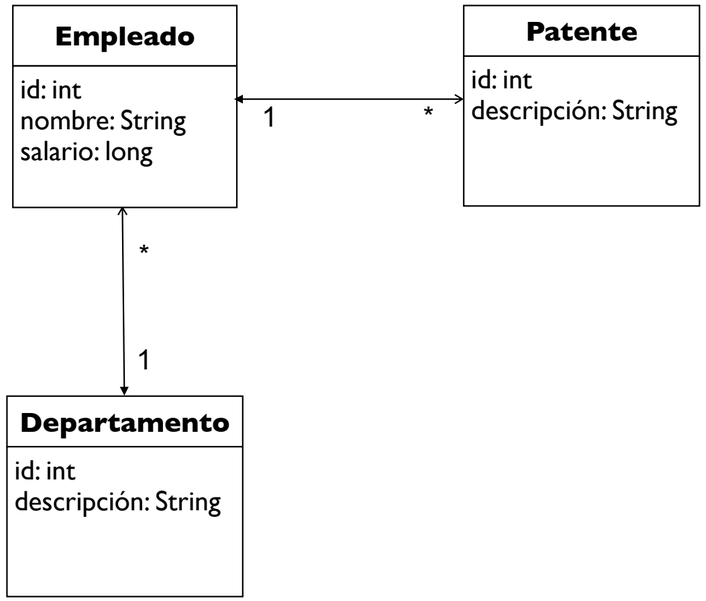
```java
@Entity
@Table(name = "PROYECTO_EMPLEADO")
public class ProyectoEmpleado {
    @Embeddable
    public static class Id implements Serializable {

        @Column(name = "PROYECTO_ID")
        private int proyectoId;

        @Column(name = "EMPLEADO_ID")
        private int empleadoId;

        public Id() {}

        public Id(int proyectoId, int empleadoId) {
            this.proyectoId = proyectoId;
            this.empleadoId = empleadoId;
        }
        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
                Id that = (Id) o;
                return this.proyectoId.equals(that.proyectoId) &&
                       this.empleadoId.equals(that.empleadoId);
            } else {
                return false;
            }
        }

        public int hashCode() {
            return proyectoId.hashCode() + empleadoId.hashCode();
        }
    }

    @EmbeddedId
    private Id id = new Id();

    @Column(name = "FECHA")
    private Date fecha = new Date();

    @Column(name = "CARGO")
    private String cargo;

    @ManyToOne
    @JoinColumn(name="PROYECTO_ID",
                insertable = false,
                updatable = false)
    private Proyecto proyecto;

    @ManyToOne
    @JoinColumn(name="EMPLEADO_ID",
                insertable = false,
                updatable = false)
    private Empleado empleado;

    …
}
```

Código (2)

```
public ProyectoEmpleado(Proyecto proyecto,
                        Empleado emplado,
                        String cargo) {
    this.proyecto = proyecto;
    this.empleado = empleado;
    this cargo = cargo;

    this.id.proyectoId = proyecto.getId();
    this.id.empleadoId = empleado.getId();

    // Garantizamos la actualización en memoria
    proyecto.getProyectoEmpleados().add(this);
    empleado.getProyectoEmpleados().add(this);
}

// Getters y setters
...
}
```

Carga perezosa

- Al recuperar un departamento sería muy costos recuperar todos sus empleados
- En las relaciones "a muchos" JPA utiliza la cara 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();
   // ...
}
```

¿Preguntas?