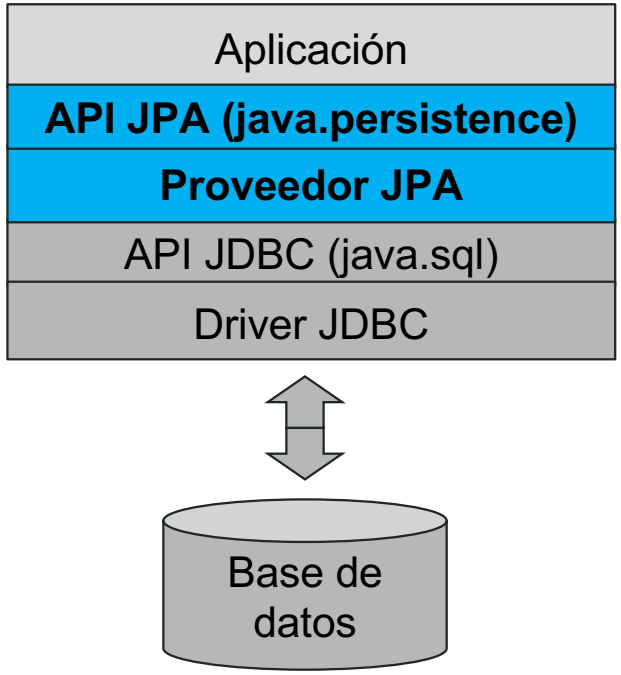
Arquitecturas Web

JPA– Anotaciones para Asociaciones y JPQL

JPA: Mapeo objeto-relacional en Java

- JPA (Java Persistence API) es una API estándar de Java para implementar mapeo objeto-relacional.
- Es una **abstracción** sobre JDBC



- JPA es solo la API. Para usarla, se necesita una **implementación o "proveedor"** de la misma, como Hibernate, EclipseLink, OpenJPA, etc.
- Se configura a través de metadatos (archivos xml y/o anotaciones)

Repaso de Programación Orientada a Objetos

- Las clases, al igual que los objetos, no existen de modo aislado. La Orientación a Objetos (POO) intenta modelar aplicaciones del mundo real tan fielmente como sea posible y por lo tanto debe reflejar estas relaciones entre clases y objetos

Existen tres clases básicas de relaciones entre los objetos:

- Asociación
- Dependencia
- Generalización / Especialización

Asociación

- Una asociación es una relación estructural que describe una conexión (de cualquier tipo) entre objetos
- Se da cuando una clase “usa” cosas de otra clase para realizar algo

Navegación de las asociaciones: representa la visibilidad entre los objetos

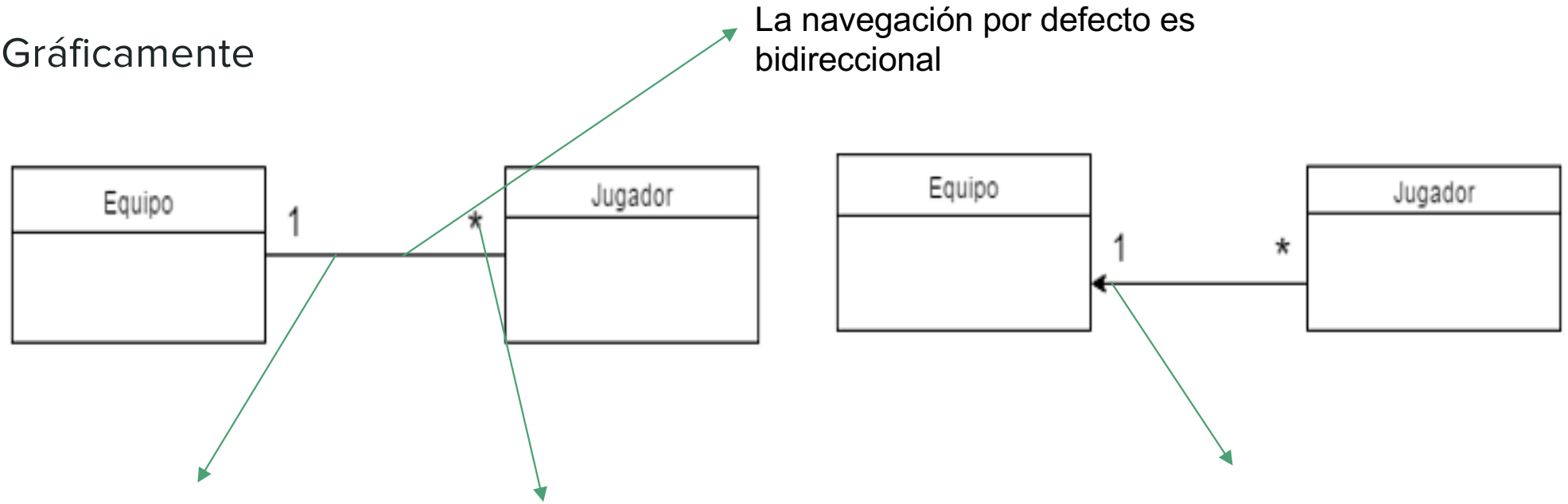
- **Uni-direccionales**: Reflejan un objeto que tiene una referencia a otro objeto
- **Bi-direccionales** : Representan dos objetos que mantienen referencias al objeto contrario

Multiplicidad: La multiplicidad de una asociación determina cuantos objetos de cada tipo intervienen en la asociación

- **uno_a_uno**
- **uno_a_muchos**
- **muchos_a_muchos**

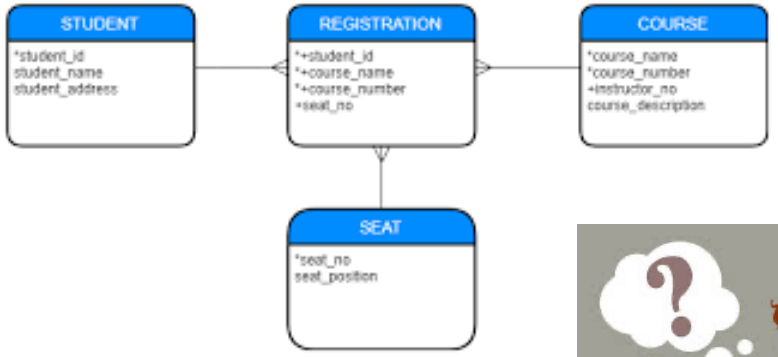
Asociación

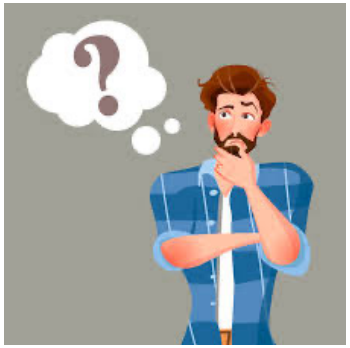
Gráficamente



Asociación en JPA

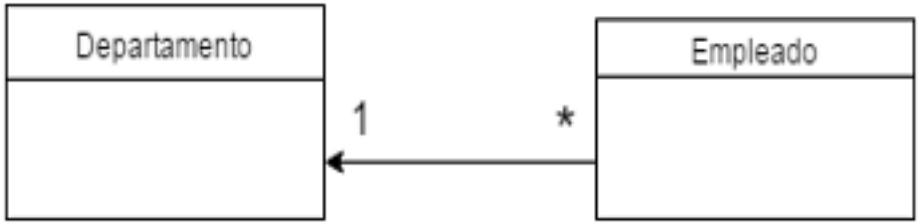
- Las posibles anotaciones en JPA para las asociaciones son:
  - **@ManyToOne**
  - **@OneToMany**
  - **@OneToOne**
  - **@ManyToMany**





Ejemplo de Asociación

Diseño en UML



- Asociación entre Empleado y Departamento
- El empleado trabaja en un departamento.
- Navegabilidad ??
- Multiplicidad ??
- Es aplicable **Many-To-One**

Ejemplo de Asociación (JPA)

Código Java, configuración: **Many-to-One (unidireccional)**

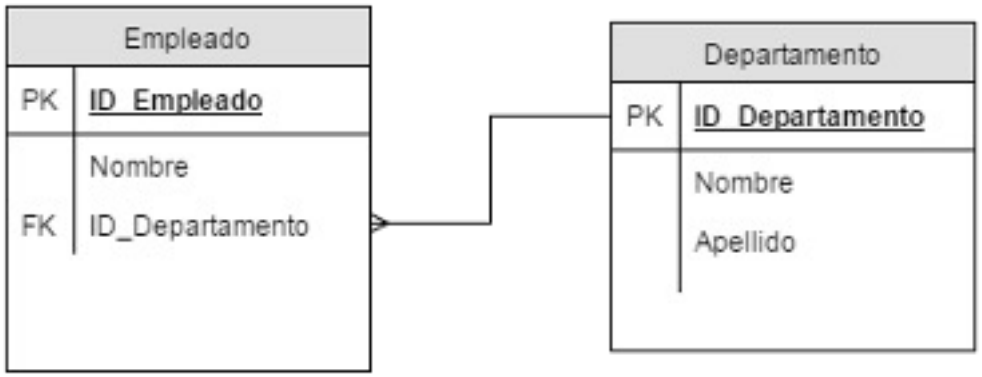
```java
@Entity
public class Empleado
{
   @Id

   @Column(name="ID_Empleado")
   private int eid;
   private String nombre;

   @ManyToOne
@JoinColumn(name="ID_Departamento")
   private Department department;
```

```java
@Entity
public class Departamento
{
   @Id

   @Column(name="ID_Departamento")
   private int id;
   private String nombre;
```

- Empleado es dueño de la relación. Tiene la FK
- Anotación `@ManyToOne`

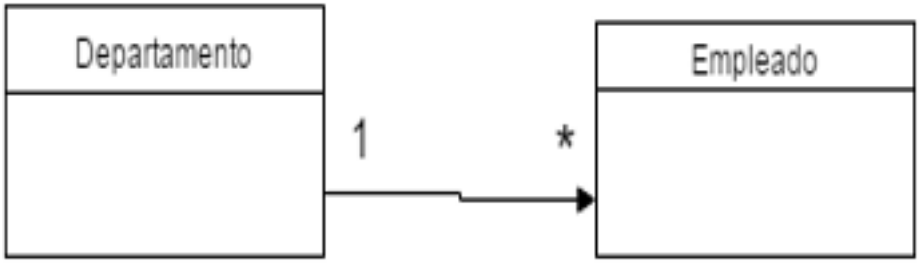
Nombre de columna particular

Resultado en la BD



Otro Ejemplo de Asociación

Diseño en UML



- Asociación entre Departamento y Empleado
- El departamento tiene muchos empleados
- Navegabilidad ??
- Multiplicidad ??
- Relación **One-To-Many**

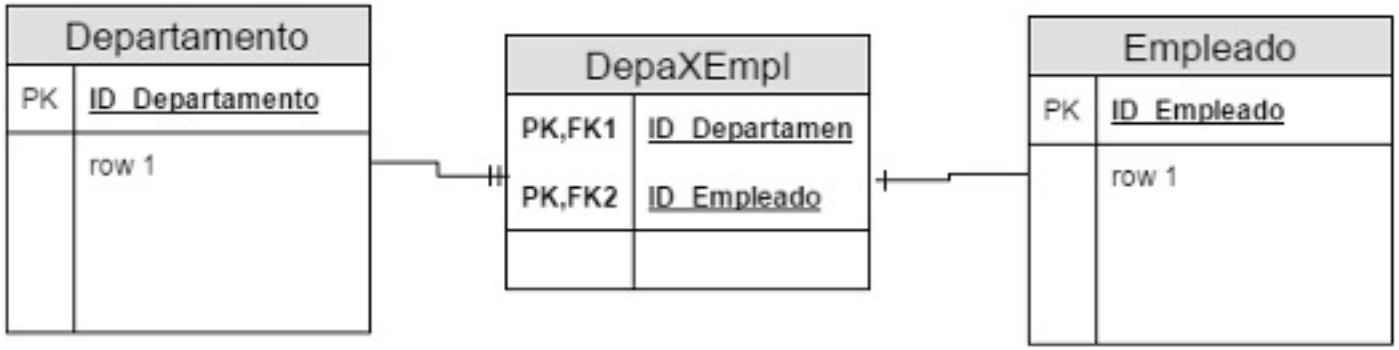
Relaciones entre entidades (JPA)

Código Java, configuración: **One-to-Many (unidireccional)**

```java
@Entity
public class Departmento {

    @Id
    private int id;
    private String name;

    @OneToMany

    private List listaEmpleado;
```

```java
@Entity
public class Empleado
{
   @Id

   @Column(name="ID_Empleado")
   private int eid;
   private String nombre;
```

- Departamento con muchas empleados
- Anotación `@OneToMany`
- JPA utilizará por defecto una tabla de unión (join table)

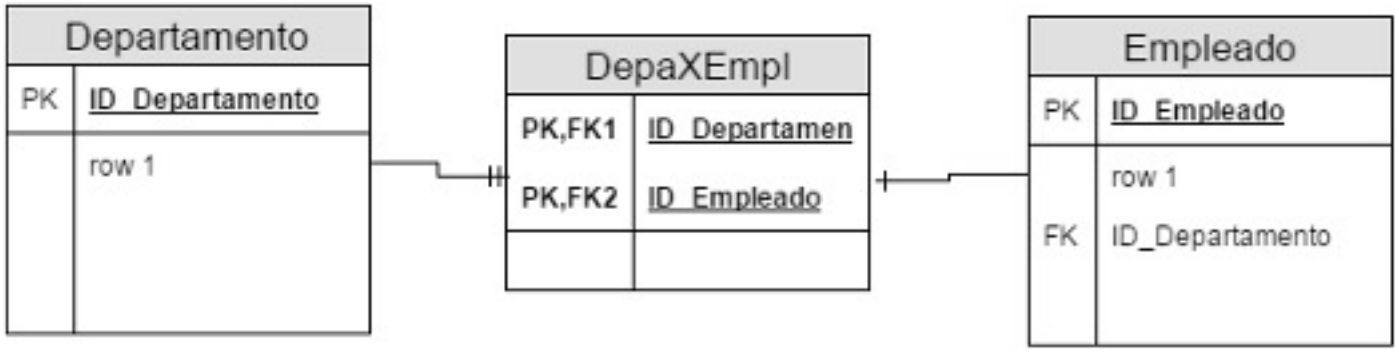


Relaciones entre entidades

Código Java, configuración: **One-to-Many y Many-to-One (bidireccional)**

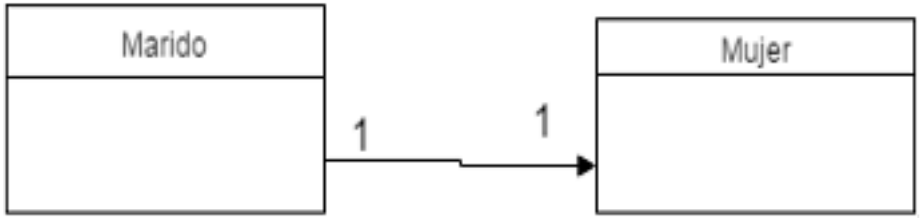
```
@Entity
public class Departmento {

    @Id
    private int id;
    private String name;

    @OneToMany

    private List listaEmpleado;
```

```
@Entity
public class Empleado
{
   @Id

   @Column(name="ID_Empleado")
   private int eid;
   private String nombre;

   @ManyToOne

@JoinColumn(name="ID_Departamento")
   private Department department;
```



Y Otro Ejemplo de Asociación

Diseño en UML



- Asociación entre Marido y Mujer
- Navegabilidad ??
- Multiplicidad ??

Relaciones entre entidades (JPA)

Código Java, configuración: **One-to-One (unidireccional)**

```
@Entity
public class Marido{
    @Id
    private int id;
    @OneToOne
    private Mujer mujer;
          // Getters y setters
}
```

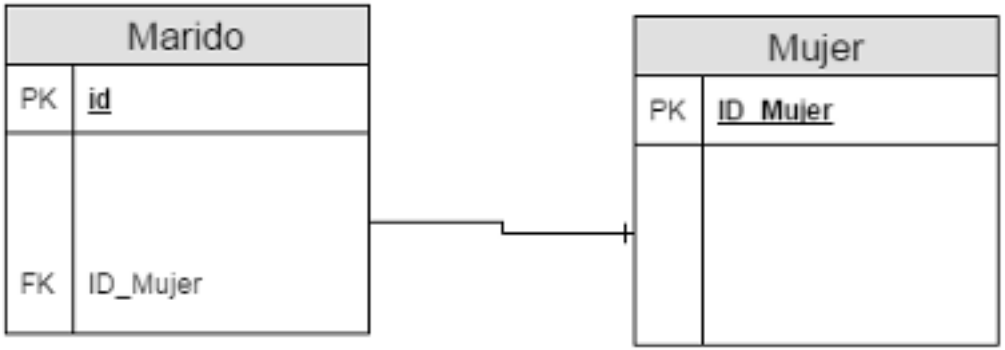
```
@Entity
public class Mujer{
    @Id
    private int id;

    // Getters y setters
}
```

- Marido es dueña de la relación. Tiene la FK
- Anotación `@OneToOne`
- En la BD la tabla Marido, tendrá una columna que representa la relación (FK)

También se puede usar:

```
@OneToOne
@JoinColumn(name = "ID_Mujer")
private Direccion direccion;
```

Para especificar una columna concreta



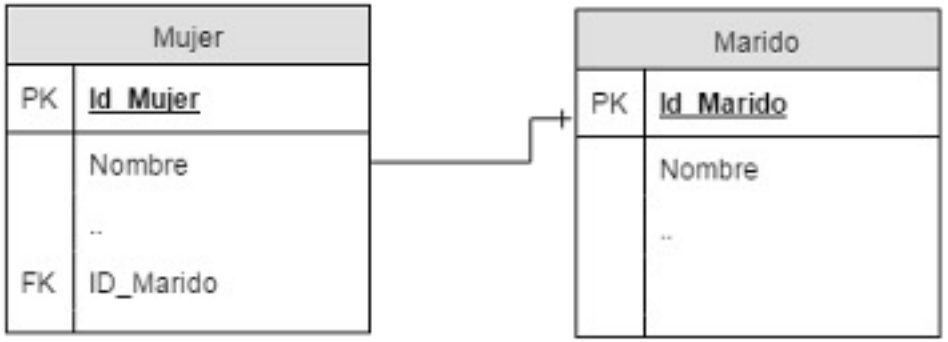
Relaciones entre entidades

Código Java, configuración: **one-to-one (bidireccional)**

```
@Entity
public class Mujer {
    @Id
    private int id;
    @OneToOne
    private Marido marido;

    // Getters y setters
}

@Entity
public class Marido {
    @Id
    private int id;
    @OneToOne(mappedBy = "marido")
    private Mujer mujer;
}
```

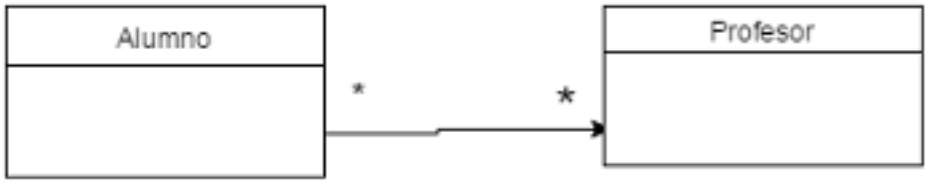
Indica que mujer es dueña de la relación

1. Ambos mantienen una referencia
2. El dueño de la relación se especifica explícitamente
3. Ambos tiene la anotación `@OneToOne`
4. Uno de ellos debe indicar que la parte contraria es la dueña de la relación



Y Otro Ejemplo de Asociación

Diseño UML



- Relación entre Alumno y Profesor
- Relación muchos a muchos
- Navegabilidad ??
- Multiplicidad ??

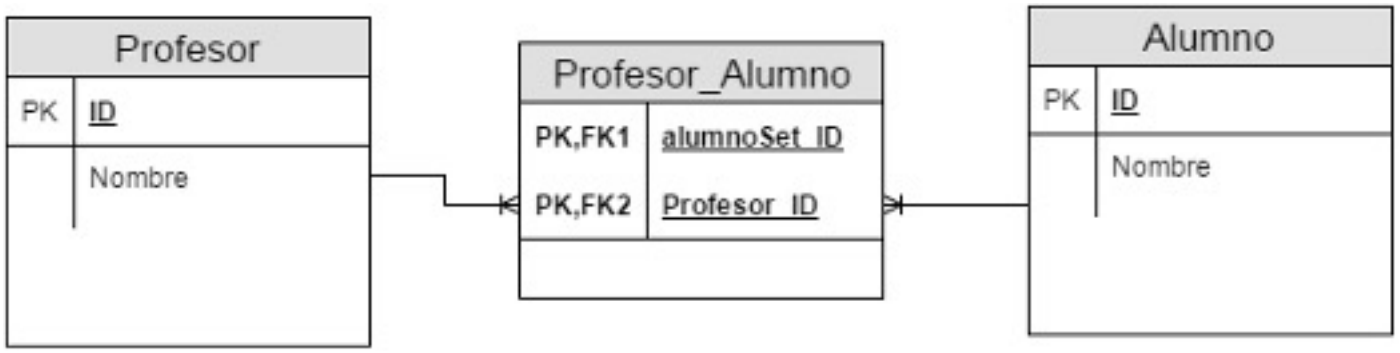
Relaciones entre entidades

Código Java, configuración: **Many-to-Many (unidireccional)**

```java
@Entity
public class Alumno
{
   @Id
   private int id;
   private String nombre;

   @ManyToMany
   private Set<Profesor> profesorSet;
```

```java
@Entity
public class Profesor
{
   @Id
   private int id;
   private String nombre;
```

Resultado en Tabla



Y Otro Ejemplo de Asociación

Diseño en UML

- Relación entre Alumno y Profesor.
- Relación muchos a muchos
- Navegabilidad ??
- Multiplicidad ??

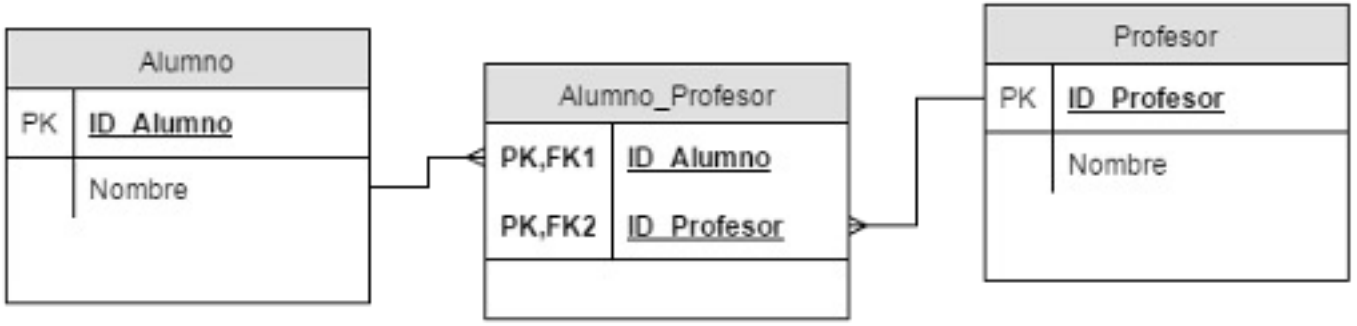
Relaciones entre Entidades

Código Java, configuración: **Many-to-Many (bidireccional)**

```java
@Entity
public class Alumno
{
   @Id
   private int id;
   private String nombre;

   @ManyToMany(mappedBy ="alumnoSet")
   private Set<Profesor> profesorSet;
```

```java
@Entity
public class Profesor
{
   @Id
   private int id;
   private String nombre;
   @ManyToMany
   private Set<Alumno> alumnoSet;
```

- Un profesor tiene muchas alumnos y un alumno tiene muchos profesores
- Dueño de la relación se especifica explícitamente (mappedBy=)
- Anotación `@ManyToMany`



Lectura temprana y Lectura demorada

- Ciertas propiedades pueden no ser necesarias en el momento de la creación del objeto
- Leer una propiedad desde la base de datos la primera vez que un cliente intenta leer su valor (lectura demorada)
- Leerla cuando la entidad que la contiene es creada (lectura temprana).
- Si la propiedad nunca es accedida, nos evitamos el costo de crearla

Lectura temprana o demorada en asociaciones

- uno-a-uno y muchos-a-uno: temprana (eager) (por defecto)
- uno-a-muchos y muchos-a-muchos: demorada (lazy) (por defecto)

```
@OneToMany(fetch = FetchType.EAGER)
private List<Pedido> pedidos;
```

Se puede modificar el tipo de lectura.

- Tener en cuenta el impacto en el rendimiento de la aplicación

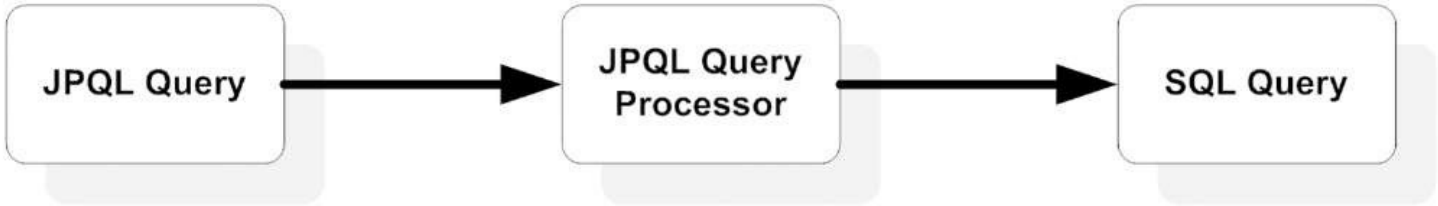
Propiedades de las anotaciones de Asociación

Propiedad	Descripción
`targetEntity`	Para especificar el tipo (clase) de la entidad a la que se hace referencia. Por defecto, infiere el tipo
`cascade`	Para especificar que las operaciones de persistencia pueden abarcar también a las referencias. Por defecto es vacío. Probar “ALL”
`fetch`	Para especificar el tipo de lectura de la referencia.
`mappedBy`	Para especificar el dueño del relación en las asociaciones bidireccionales. (Many-to-one, no tiene esta propiedad)

Arquitecturas Web

Java Persistence Query Language (JPQL)

JPQL



- Lenguaje de consulta de persistencia en Java (JPA)
- JPQL permite realizar consultas en base a multitud de criterios (como por ejemplo: el valor de una propiedad, o condiciones booleanas), y obtener más de un objeto por consulta
- JPQL define consultas para las entidades y su estado persistente
- JPQL permite escribir consultas **portables** que funcionan con independencia de la BD y su dialecto SQL particular
- El lenguaje de las consultas es muy parecido a SQL

JPQL – Consultas comunes

```
1. SELECT p FROM Pelicula p
2. SELECT p FROM Pelicula p
   WHERE p.duracion < 120
3. SELECT p FROM Pelicula p
   WHERE p.duracion BETWEEN
   90 AND 150
4. UPDATE Articulo a
   SET a.descuento = 15
5. DELETE FROM Pelicula p
   WHERE p.duracion > 190
6. SELECT COUNT(p) FROM
   Pelicula p
```

- Las palabras *SELECT* y *FROM* tienen un significado similar a las sentencias homónimas del lenguaje SQL, indicando que se quiere seleccionar (SELECT) cierta información desde (FROM) cierto lugar
- La segunda *p* es un alias para la clase *Pelicula*, y ese alias es usado por la primera *p* (llamada expresión) para acceder a la clase (tabla) a la que hace referencia el alias, o a sus propiedades (columnas)
- El alias *p* permite utilizar la expresión *p.titulo* para obtener los títulos de todas las películas almacenadas en la base de datos
- Para consultas condicionales, se usa WHERE
- Operadores: >,<, =, *AND, OR, NOT,BETWEEN*
- Operaciones de actualización
- Operaciones de borrado
- Funciones agregadas: *AVG, COUNT, MAX, MIN, SUM*

JPQL – Consultas con entidades relacionadas

Una lista como referencia	

`SELECT c`
` FROM Cliente c JOIN c.ordenes o`
` WHERE c.estado = 1 AND o.precioTotal > 10000`	

`SELECT c`
` FROM Cliente c, IN(c.ordenes) o`
` WHERE c.estado = 1 AND o.precioTotal > 10000`
	Identifica un solo miembro de la colección.
	No se puede hacer c.ordenes.estado

`SELECT e`
`FROM Equipo e, IN(e.jugadores) j`
`WHERE j.direccion.ciudad = :ciudad`
	Relaciones múltiples

Una referencia simple	

`SELECT c`
` FROM Cliente c JOIN c.direccion d`
` WHERE d.calle = 'calle1'`	

JPQL – Consultas con parámetros

Parámetros con nombre:
Se usa “:”, luego del objeto Query se usa setParameter()

```
em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .getResultList();
```

Parámetros con posición:
Se usa “?#”, donde “#” es la posición del parámetro en la consulta

```
em.createQuery(
        “SELECT c FROM Customer c WHERE c.name LIKE ?1”)
        .setParameter(1, name)
        .getResultList();
```

JPQL – Ejecución de sentencias

- El lenguaje JPQL puede integrarse a través de implementaciones de la interfaz Query
- Dichas implementaciones se obtienen a través del EntityManager

- Los más usados son:
  - `createQuery(String jpql)`
  - `createNamedQuery(String name)`
  - `createNativeQuery(String sql)`

JPQL: createQuery

```
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
                .createEntityManagerFactory("introduccionJPA");
        EntityManager em = emf.createEntityManager();

        String jpql = "SELECT a FROM Alumno a";
        Query query = em.createQuery(jpql);
        List<Alumno> resultados = query.getResultList();
        for(Alumno  p : resultados) {
            System.out.println(p.getNombre());
        }

        em.close();
        emf.close();
    }
```

```
String jpql = "SELECT p FROM Pelicula p WHERE p.duracion > ?1 AND
p.genero = ?2"
Query query = em.createQuery(jpql);
query.setParameter(1, 180);
query.setParameter(2, "Accion");
List<Pelicula> resultados = query.getResultList();
```

- Se obtiene una implementación de `Query` mediante el método `createQuery(String)` de `EntityManager`
- `getResultList()` retorna un objeto `List` con todas las entidades devueltas por la sentencia JPQL
- Esta sentencia es “dinámica”, ya que es generada cada vez que se ejecuta
- Ejecución de sentencias con parámetros dinámicos. mediante el método `setParameter()`

JPQL: createNamedQuery

```
@Entity
@NamedQuery(name=Alumno.BUSCAR_TODOS, query="SELECT a FROM
Alumno a")
public class Alumno{
    public static final String BUSCAR_TODOS =
"Alumno.buscarTodos";
    // ...
}
```

```
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
                .createEntityManagerFactory("introduccionJPA");
        EntityManager em = emf.createEntityManager();

        Query query = em.createNamedQuery(Alumno.BUSCAR_TODOS);
        List<Alumno> resultados = query.getResultList();
        for(Alumno  p : resultados) {
            System.out.println(p.getNombre());
        }

        em.close();
        emf.close();
    }
```

- Una vez definidos, no pueden ser modificados
- Son leídos y transformados en sentencias SQL cuando el programa arranca por primera vez
- Las consultas con nombre son definidas mediante metadatos (anotación `@NamedQuery`)
- El nombre de la consulta debe ser único dentro de su unidad de persistencia
- Es una buena idea utilizar una constante definida dentro de la propia entidad, y usarla como nombre de la consulta

JPQL: createNativeQuery

```
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
.createEntityManagerFactory("introduccionJPA");
        EntityManager em =
emf.createEntityManager();

        Query query = em.createNativeQuery("Select
* from Profesor",Profesor.class);
        List<Profesor> resultados =
query.getResultList();
        for(Profesor  p : resultados) {
            System.out.println(p.getNombre());
        }

        em.close();
        emf.close();
    }
```

- A veces se requiere una sentencia SQL **nativa** en lugar de una sentencia JPQL
- Las consultas SQL nativas pueden ser definidas de manera estática como se hace con las consultas con nombre, obteniendo los mismos beneficios de eficiencia y rendimiento
- Tanto las consultas con nombre como las consultas nativas SQL estáticas son muy útiles para definir consultas inmutables (por ej., buscar todas las instancias de una entidad en la base de datos)
- Las candidatas son aquellas consultas que se mantienen inmutables entre distintas ejecuciones del programa (sin parámetros dinámicos), y que son usadas frecuentemente

JPQL: TypedQuery

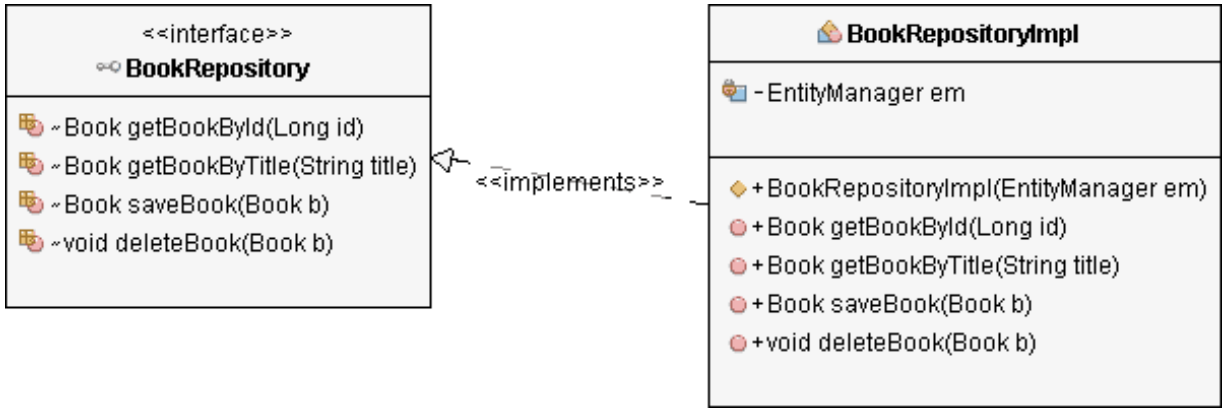
```
public UserEntity getUserByIdWithTypedQuery(Long id) {

    TypedQuery<UserEntity> typedQuery = getEntityManager()
           .createQuery("SELECT u FROM UserEntity u WHERE u.id=:id",UserEntity.class);

    typedQuery.setParameter("id", id);
    return typedQuery.getSingleResult();

}
```

- Sigue siendo JPQL
- Cuando se conoce el tipo de retorno
- Y es más confiable en términos de los resultados

El patron Repository

- Es una abstracción del concepto de repositorio
- Permite agrupar las operaciones lógicas de read/write de entidad(es) específica(s)
- La lógica de negocio accede al repositorio, independientemente de su implementación

- La implementación puede basarse en JPA, JPQL, etc.



Ejemplo de Repository

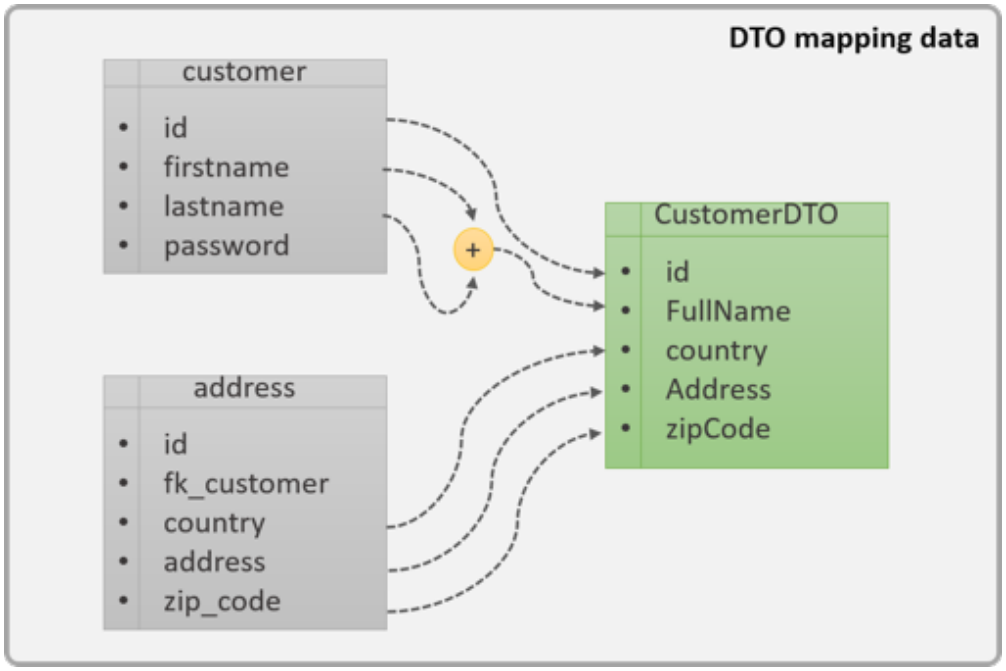
```
public interface BookRepository {
    Book getBookById(Long id);
    Book getBookByTitle(String title);
    Book saveBook(Book b);
    void deleteBook(Book b);
}
```

```
public class BookRepositoryImpl implements BookRepository {
    private EntityManager em;
    public BookRepositoryImpl(EntityManager em) {
        this.em = em;
    }
    public Book getBookById(Long id) {
        return em.find(Book.class, id);
    }
    public Book getBookByTitle(String title) {
        TypedQuery<Book> q = em.createQuery("SELECT b
        FROM Book b WHERE b.title = :title", Book.class);
        q.setParameter("title", title);
        return q.getSingleResult();
    }
    ...
```

```
    ...
    public Book saveBook(Book b) {
        if (b.getId() == null) {
            em.persist(b);
        } else {
            b = em.merge(b);
        }
        return b;
    }
    public void deleteBook(Book b) {
        if (em.contains(b)) {
            em.remove(b);
        } else {
            em.merge(b);
        }
    }
}
```

Patrón DTO (Data Transfer Object)

- En un backend Java, normalmente se tiene una capa de servicios o presentación, que “expone” datos a los clientes
- Por distintas razones, no se quiere exponer directamente las entidades (y sus datos), sino un **“resumen” de una o varias entidades (o fuentes)**



  - Seguridad, separación de intereses, performance, reducir invocaciones, etc.
  - Normalmente, un DTO es una entidad de transferencia de datos, de solo lectura

DTO EN JPA

```
@Entity
@Table(name="customers")
public class Customer {
  @Id
  @GeneratedValue(strategy=
  GenerationType.IDENTITY)
  private Long id;
  @Column(name="firstname")
  private String firstname;
  @Column(name="lastname")
  private String lastname;
  @Column(name="password")
  private String password;

/** GET and SET */
}
```

```
@Entity
@Table(name="address")
public class Address {
  @Id
  @GeneratedValue(strategy=
  GenerationType.IDENTITY)
  private Long id;
  @ManyToMany()
  @JoinColumn(name="fk_customer")
  private Customer customer;
  @Column(name="country")
  private String country;
  @Column(name="address")
  private String address;
  @Column(name="zipcode")
  private String zipCode;
/** GET and SET */
}
```

```
public class CustomerDTO
implements Serializable{
  private Long id;
  private String FullName;
  private String country;
  private String Address;
  private String zipCode;
  /** GET and SET */
}
```

DTO en un Servicio

```
@Path("customers")
public class CustomerService {
  @GET
  @PathParam("{customerId}")
  private Response findCustomer(@PathParam("customerId") Long customerId) {
    Customer customer = customerDAO.findCustomerById(customerId); //Entity
    Address address = addressDAO.findAddressByCustomer(customerId); //Entity
    //Create dto
    CustomerDTO dto = new CustomerDTO();
    dto.setAddress(address.getAddress());
    dto.setCountry(address.getCountry());
    dto.setZipCode(address.getZipCode());
    dto.setFullName(customer.getFirstname() + " " + customer.getLastname());
    dto.setId(customer.getId());
    //Return DTO
    return Response.ok(dto, MediaType.APPLICATION_JSON).build();
  }
}
```

DTO en una consulta JPA

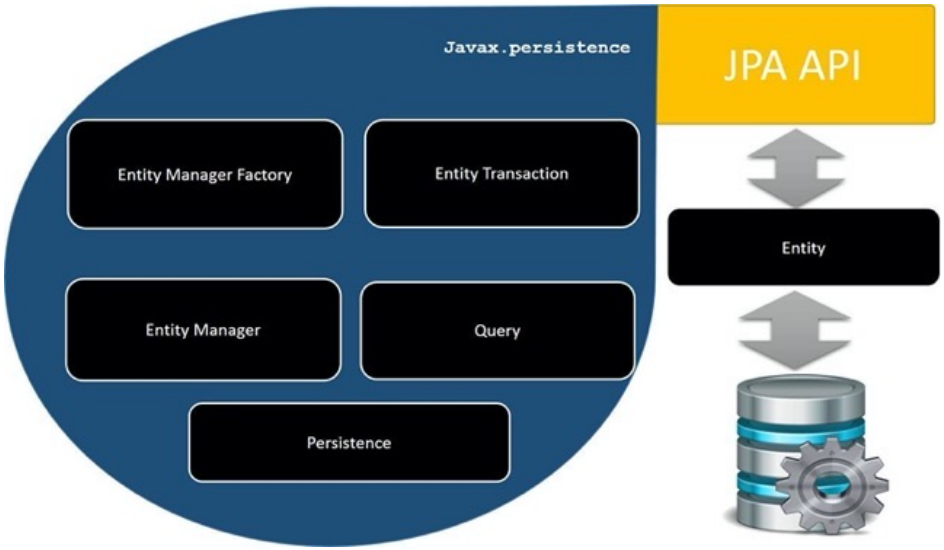
```
List<PostDTO> postDTOs =
entityManager.createQuery("
    SELECT new PostDTO(
       p.id,
       p.title
    )
    from Post p
    where p.createdOn > :fromTimestamp
    ", PostDTO.class)
.setParameter(
   "fromTimestamp",
    Timestamp.from(LocalDate.of(2020, 1, 1).atStartOfDay().toInstant(ZoneOffset.UTC))
).getResultList();
```

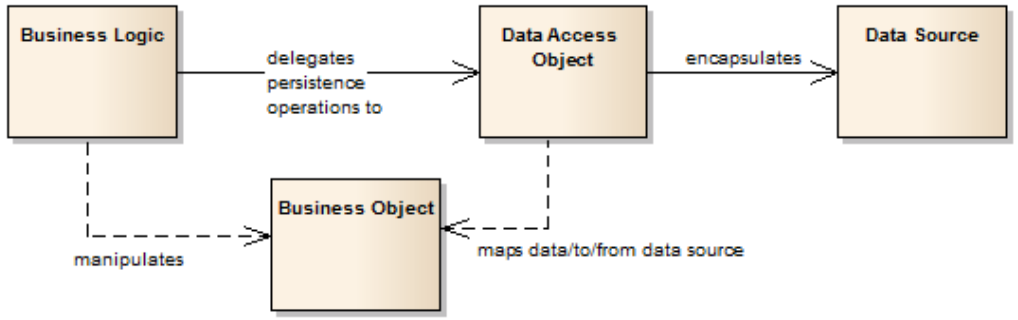
```
public class PostDTO {
    private Long id;
    private String title;
    public PostDTO(Number id, String title) {
        this.id = id.longValue();
        this.title = title;
    }
    public Long getId() {
        return id;
    }
    public String getTitle() {
        return title;
    }
}
```

Algunos anti-patrones

- Sesiones que se abren en un lado y se cierran en otro lado “lejos” de la apertura
- Resolver toda la complejidad de una consulta en Java (y en memoria), en vez de hacerlo con consultas específicas en JPQL (eficiencia) … o en SQL
- Gestión de excepciones
- Definición de ámbito/granularidad de las transacciones
- …

¿Realmente hacen falta los DAOs?





Algunas referencias

http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html

http://openjpa.apache.org/builds/1.2.3/apache-openjpa/docs/jpa_langref.html