



# **Persistencia de objetos mediante un ORM**

**JPA - Hibernate**

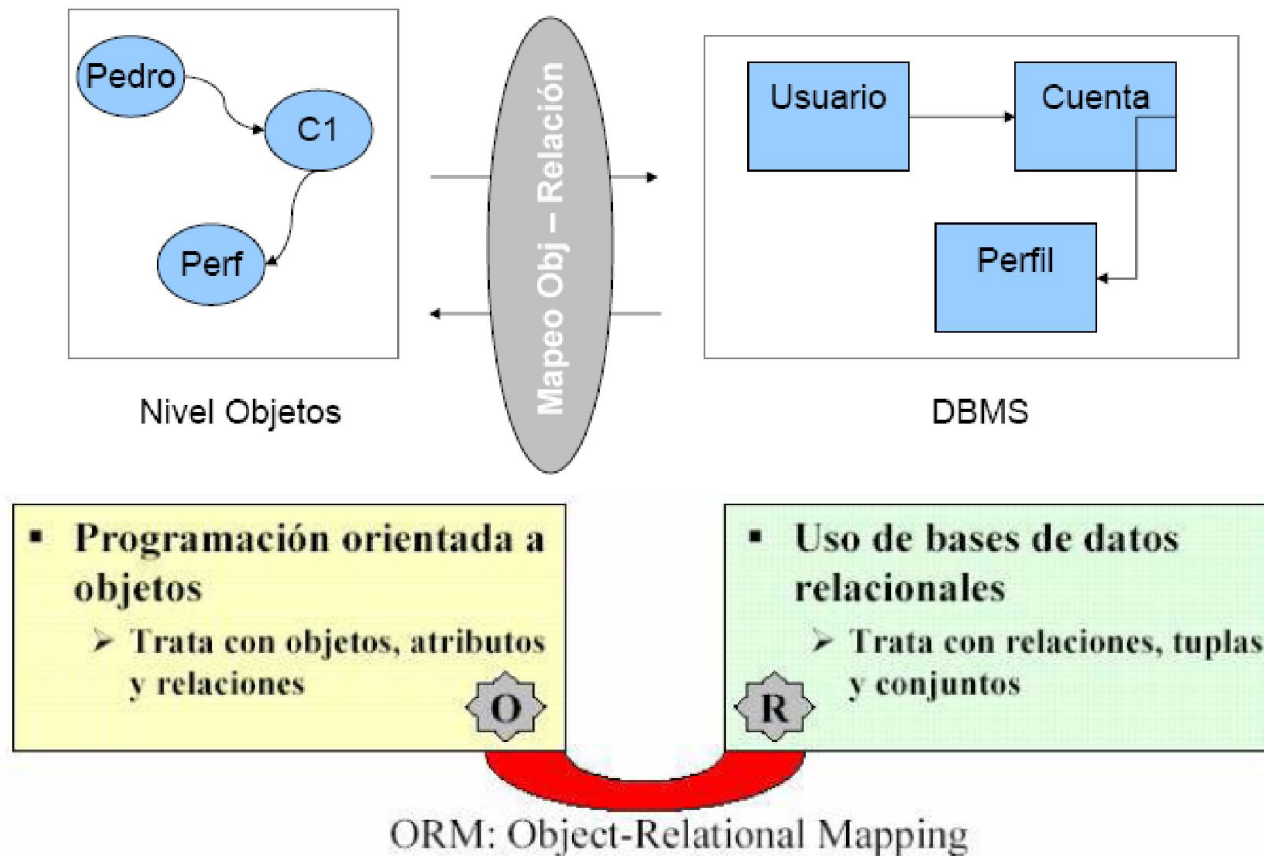


# ¿Qué es un ORM?

## ORM - Object-Relational Mapping

Es una técnica de programación que **nos permite vincular los objetos** usados en nuestro modelo de la aplicación **con una base de datos relacional**.

# ¿Por qué usar un ORM?



**Problema:** buena parte del código de una aplicación se produce para realizar la correspondencia  $O \leftrightarrow R$

**Solución:** utilizar un ORM, por ejemplo Hibernate JPA



# ¿Cuáles son las ventajas y desventajas de usar un ORM?

## Ventajas

- **Permite concentrarnos en el diseño de objetos** sin pensar demasiado en la forma en la que lo persistiremos
- **Facilidad y velocidad de uso**, ya que la mayoría provee creación automática del esquema de base de datos y operaciones de alto nivel para CRUD
- **Evita el “coding” repetitivo** de sentencias DDL y DML
- **Detección automática de cambios** y persistencia por alcance
- **Abstracción del motor de base de datos**
- **Portabilidad**, es posible el cambiar a otra base de datos, incluso durante el desarrollo de un proyecto



# ¿Cuáles son las ventajas y desventajas de usar un ORM?

## Desventajas

- **Menor rendimiento**, por ejemplo para una consulta el sistema tendrá que convertir la consulta al SQL del proveedor de BD utilizado
- **Curva de aprendizaje**, los frameworks ORM suelen tener mucha funcionalidad así que llegar a explotar su máximo rendimiento costará cierto tiempo



# ¿Qué puedo utilizar como ORM en Java?

## **JPA - Java Persistence API**

El JCP desarrolló una API de persistencia llamada **JPA**, para la plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE).

Fue incluida como parte de la especificación EJB 3.0



# JPA y su relación con Hibernate

- **Framework Hibernate Nativo (con mapeos en XML)**

- La persistencia se maneja a través de objetos **Session** creados por un **SessionFactory**. La configuración se encuentra en **hibernate.properties** o **hibernate.cfg.xml**
- Cada clase necesita ser mapeada con un archivo de configuración, por ejemplo

```
<mapping resource="modelo/Usuario.hbm.xml"/>
<mapping resource="modelo/Productos.hbm.xml"/>
```

- **Framework Hibernate con Anotaciones**

- Aparece cuando surgieron las anotaciones en Java (versión 1.5)
- Las anotaciones de Hibernate reemplazan a los mapeos XML de Hibernate
- Se debe incorporar en el proyecto el módulo **Annotations**
- Solo se declaran las clases anotadas, por ejemplo `<mapping class="modelo.Usuario"/>`

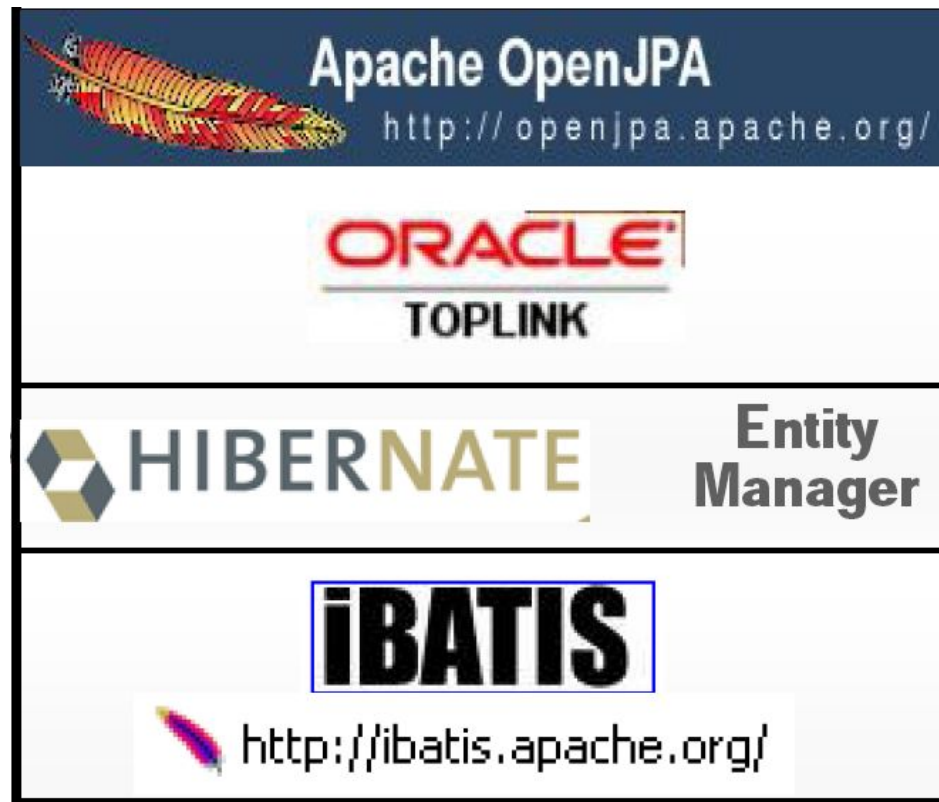
- **Especificación JPA - Java Persistence API**

- **Hibernate implementa JPA**
- La persistencia se maneja a través de objetos **EntityManager** creados por un **EntityManagerFactory** (EMF). La configuración se encuentra en el archivo **persistence.xml**
- En el archivo **persistence.xml** se define una **unidad de persistencia**, allí se declaran las clases anotadas y las propiedades del proveedor de JPA



# Motores de persistencia compatibles con JPA

Estas son algunas implementaciones disponibles:







# JPA - Java Persistence API

- Dos principios importantes de JPA:
  - ✓ El motor JPA debe ser “*pluggable*”. Se debe poder reemplazar un producto por otro sin hacer modificaciones de código.
  - ✓ El motor JPA debe poder correr fuera de un contenedor EJB 3.0/servidor JEE.

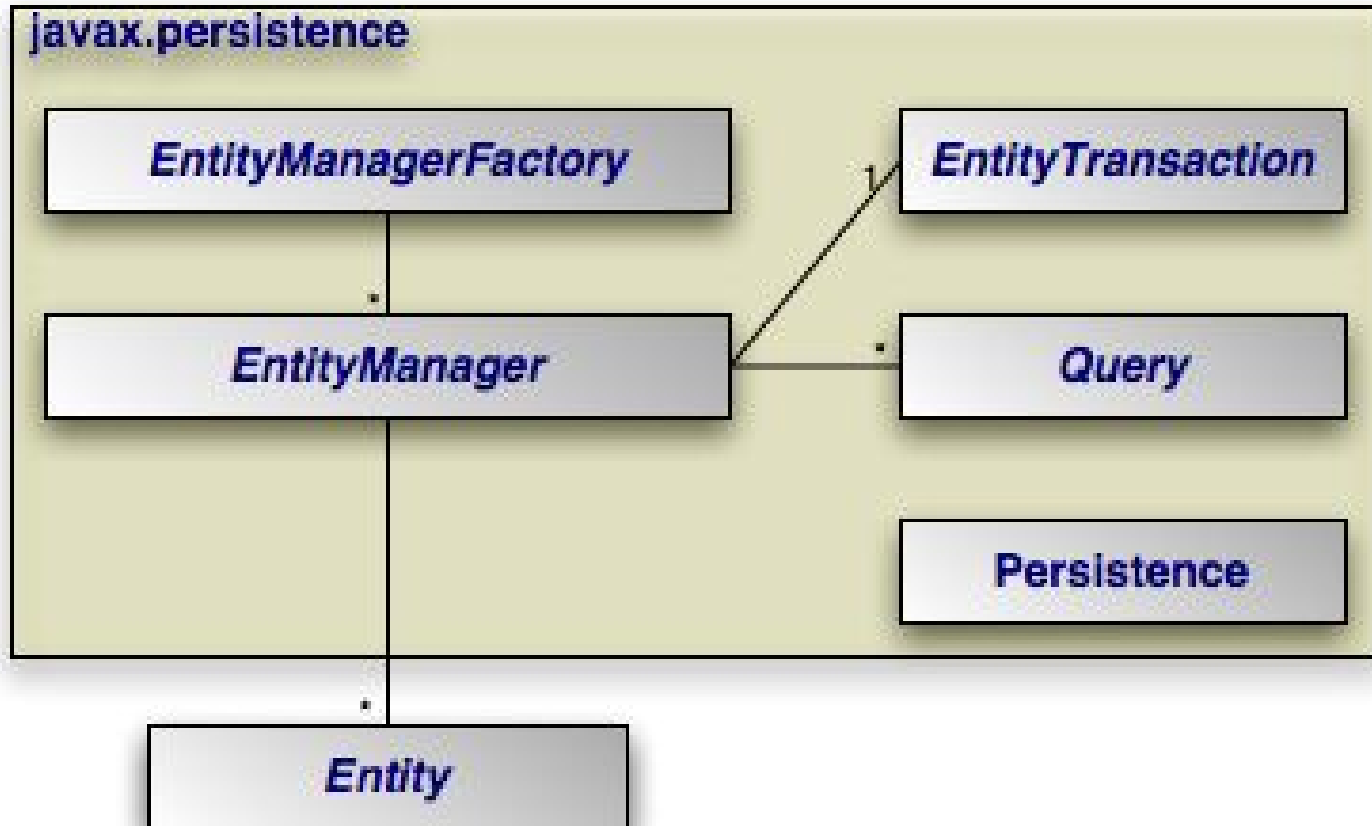


# ¿Qué provee JPA?

- Provee una interfase común de ORM (mapeo objeto-relacional) para plataformas JSE y JEE
- La persistencia estándar de Java contempla tres áreas:
  - La API llamada JPA, definida en el paquete **javax.persistence**
  - Un lenguaje de consultas **JPQL** - Java Persistence Query Language
  - Metadatos objeto/relacional



# JPA – Interfaces de la API



# JPA – Interfaces de la API

## 1 EntityManager

- clear()
- close()
- contains(Object)
- createNamedQuery(String)
- createNativeQuery(String)
- createNativeQuery(String, Class)
- createNativeQuery(String, String)
- createQuery(String)
- find(Class<T>, Object) <T>
- flush()
- getDelegate()
- getFlushMode()
- getReference(Class<T>, Object) <T>
- getTransaction()
- isOpen()
- joinTransaction()
- lock(Object, LockModeType)
- merge(T) <T>
- persist(Object)
- refresh(Object)
- remove(Object)
- setFlushMode(FlushModeType)

## 1 EntityManagerFactory

- close()
- createEntityManager()
- createEntityManager(Map)
- isOpen()

## 1 EntityTransaction

- begin()
- commit()
- getRollbackOnly()
- isActive()
- rollback()
- setRollbackOnly()

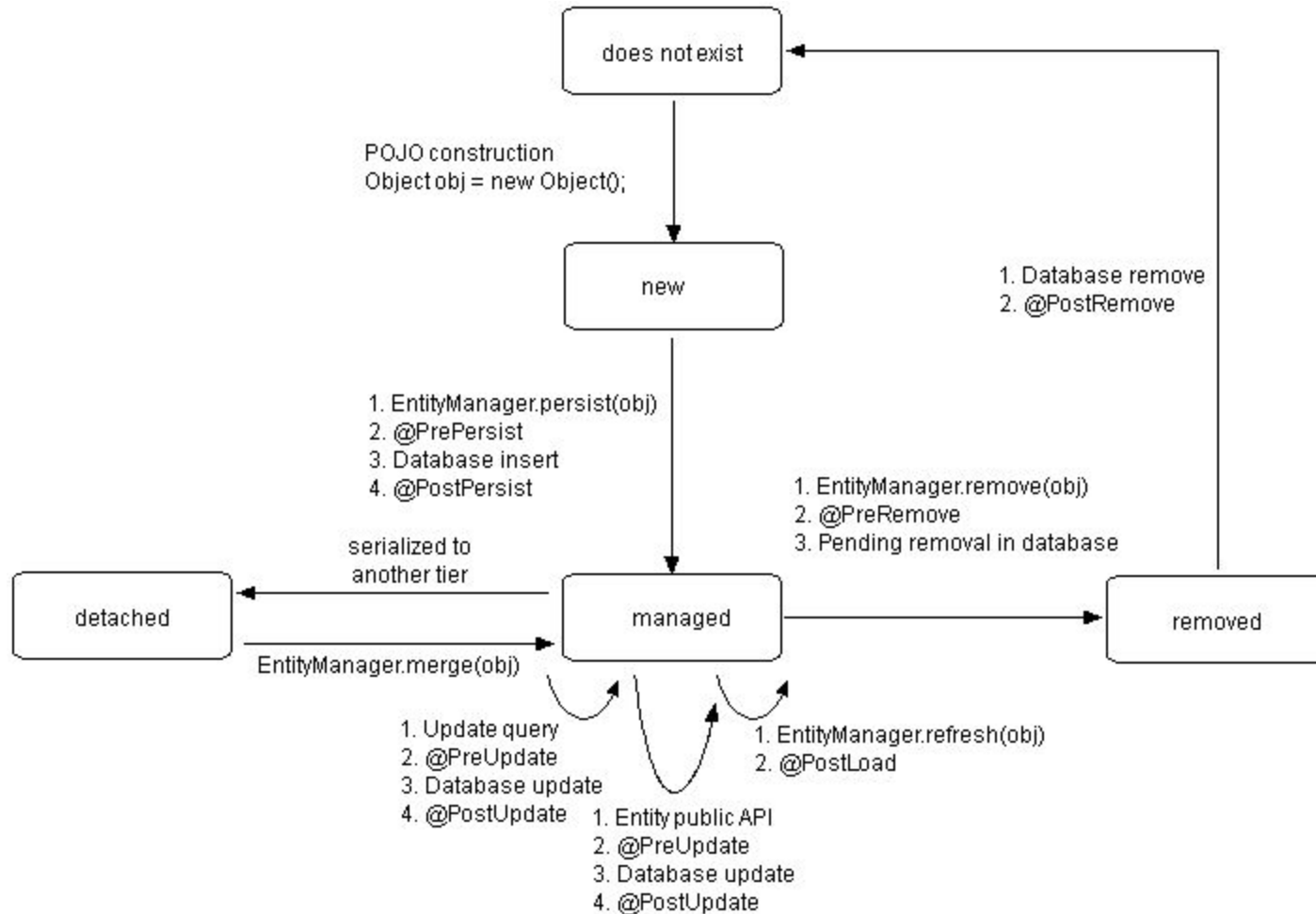
## 1 Query

- executeUpdate()
- getResultList()
- getSingleResult()
- setFirstResult(int)
- setFlushMode(FlushModeType)
- setHint(String, Object)
- setMaxResults(int)
- setParameter(int, Object)
- setParameter(int, Calendar, TemporalType)
- setParameter(int, Date, TemporalType)
- setParameter(String, Object)
- setParameter(String, Calendar, TemporalType)
- setParameter(String, Date, TemporalType)



# JPA - Ciclo de vida

## Ciclo de vida de los objetos persistentes





# JPA - Ciclo de vida

- Se gestiona desde un **EntityManager**
  - Es el gestor de persistencia de JPA
- El **EntityManager** (la sesión) es el contexto de persistencia
  - El ciclo de vida tiene lugar en la memoria de la JVM
  - Un objeto “está en sesión” cuando está en estado **Managed**
- La sesión es una caché que:
  - Garantiza la identidad java y la identidad DB
  - No habrá varios objetos en sesión representando la misma fila
  - JPA (hibernate) optimiza el SQL para minimizar tráfico a la BD



# JPA - Estados de persistencia

- **New**
  - Objetos que no tienen identidad persistente y no se asociaron con una sesión (solo existen en memoria de la JVM)
- **Managed**
  - Objeto enlazado con la sesión
  - Tienen identidad persistente
  - Todos los cambios que se le hagan serán persistentes
- **Detached**
  - Objeto persistente que sigue en memoria después de que termina la sesión: existe en java y en la BD
  - Tiene identidad persistente
- **Removed**
  - Objeto marcado para ser eliminado de la BD: existe en java y se borrará de la BD al terminar la sesión



# JPA – Configuración

Para definir una unidad de persistencia se debe agregar el archivo **persistence.xml** en el directorio **WEB-INF/classes/META-INF** de la aplicación.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  version="2.1">

  <persistence-unit name="miUP">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>mipaquete.Mensaje</class>
    <properties>
      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.password" value="admin"/>
      <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/mibd"/>
      <property name="hibernate.connection.username" value="root" />
      <property name="hibernate.default_schema" value="stock" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
    </properties>
  </persistence-unit>
```





# JPA – Clases a persistir

## Las clases a persistir son clases POJO (Plain Old Java Object)

```
public class Mensaje implements java.io.Serializable {    ..implementa esta interface (no es obligatorio)
    private Long id;
    private String texto;
    private Date dia;

    public Mensaje() {}    ..el atributo id permite identificar al objeto persistente, es como
                        la clave primaria. Si dos objetos Mensaje tienen el mismo id,
                        representan la misma fila en la Tabla.

    public Long getId(){
        return id;
    }    ..debe proveer un constructor sin
                        argumentos (requisito obligatorio para
                        Hibernate)

    private void setId(Long id){
        this.id = id;
    }

    public String getTexto(){
        return texto;
    }
    public void setTexto(String texto){
        this.texto=texto;
    }
    ...
}
```

Los campos clave deben ser de tipos primitivos como *int*, *long*, etc; clases wrapper como *Integer*, *Long*, etc; *java.lang.String*, *java.util.Date* o *java.sql.Date*



# JPA - Creación de entidades

```
package modelo;
```

```
import javax.persistence.*;
```

```
@Entity
```

```
@Table(name="MENSAJES")
```

```
public class Mensaje {
```

```
    @Id @GeneratedValue
```

```
    @Column(name="MENSAJE_ID")
```

```
    private Long id;
```

```
    private String text;
```

```
    private Date dia;
```

```
    public Mensaje(){ }
```

```
    public String getId() {  
        return id;  
    }
```

```
    public void setId(Long id) {  
        this.id = id;  
    }
```

```
    ...
```

```
}
```

..dentro de este paquete están todas las anotaciones de JPA, que se necesitan para mapear una clase java con una tabla en la base de datos.

..todas las clases anotadas con **@Entity**, tiene sus propiedades persistentes por defecto. En este ejemplo, las anotaciones **@Table** y **@Column** especifican un valor distinto en la tabla y el nombre de la columna, pero no hace falta para indicar persistencia.

..todas las entidades tienen que poseer una identidad que las diferencie del resto, por lo que deben contener una propiedad marcada con la anotación **@Id**.



# Ejemplo de persistencia y consulta

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("miUP");  
EntityManager em = emf.createEntityManager();
```

```
EntityTransaction etx = em.getTransaction();  
etx.begin();
```

```
//Creo un mensaje y lo persisto
```

```
Mensaje m = new Mensaje();  
m.setTexto("Soy un EJB3.0");  
m.setDia(new java.util.Date());
```

```
em.persist(m);
```

```
//si ahora hiciéramos m.setTexto("Otro texto"); este cambio también sería persistido
```

```
//Obtengo todos los mensajes
```

```
List<Mensaje> mensajes=(List<Mensaje>)(em.createQuery("from modelo.Mensaje m order by  
m.texto asc")).getResultList();  
for (Mensaje men:mensajes) {System.out.println("Mensaje : " +men.getTexto());}
```

```
etx.commit();  
em.close();
```



# JPQL - El lenguaje de consultas de JPA

- Para hacer las consultas en JPA se emplea un lenguaje denominado JPQL.
- JPQL es un lenguaje independiente de la plataforma y orientado a objetos.
- Está fuertemente inspirado en SQL y sus consultas se asemejan a la sintaxis de las consultas SQL
- Opera contra objetos entidad JPA en lugar de hacerlo directamente con las tablas de base de datos.

Sintaxis:

```
SELECT [<resultado>]  
  [FROM <entidad(es)>]  
  [WHERE <filtro>]  
  [GROUP BY <agrupación>]  
  [HAVING <condición>]  
  [ORDER BY <orden>]
```



# JPQL - Parámetros de Entrada

- Parámetros con nombre:

```
Query q = em.createQuery("SELECT p FROM Persona p WHERE  
    p.apellido = :ape AND o.nombre = :nom");  
q.setParameter("ape", "Perez");  
q.setParameter("nom", "Juan");
```

- Parámetros numerados:

```
Query q em.createQuery("SELECT p FROM Persona p WHERE  
    p.apellido = ?1 AND p.nombre = ?2");  
q.setParameter(1, "Perez");  
q.setParameter(2, "Juan");
```



# JPQL – Ejecución de consultas

Hay dos maneras de ejecutar una consulta JPQL:

- **getResultList**, cuando el conjunto de valores devuelto es una lista de valores, por ejemplo un SELECT de varios campos.

List results = query.**getResultList()**;

- **getSingleResult**, cuando solo se devuelve un único objeto (fila).

Object resultado = query.**getSingleResult()**;

*Puede disparar una excepción **NoResultException** si la consulta no retorna un resultado y **NonUniqueResultException** si la consulta retorna más de un resultado*



# JPA – Asociaciones

En Java y JPA todas las relaciones son *unidireccionales*.

En el mundo de BD las relaciones son *bidireccionales*.

Para implementar las asociaciones primero debemos determinar la **navegabilidad** y la **cardinalidad**.

## Navegabilidad

- En UML se establece mediante puntas de flecha en el extremo apropiado de la asociación, la ausencia de puntas de flecha indica navegabilidad en ambos sentidos.
- Determina que partes de una asociación deben ser implementadas.

## Cardinalidad

- Existen cuatro formas de “mapeo”:
  - Relación **One-To-One** (mediante valores simples).
  - Relación **Many-To-One** (mediante valores simples)
  - Relación **One-To-Many** (mediante colecciones)
  - Relación **Many-To-Many** (mediante colecciones)



# JPA – Asociaciones One-To-One

```
@Entity
public class Persona {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    private String nombre;

    @OneToOne(optional = false)
    private Documento documento;

    //...
}
```

```
@Entity
public class Documento {

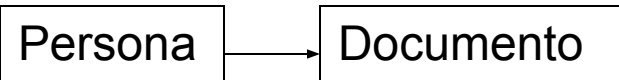
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    @Column(nullable=false)
    private String numero;

    @OneToOne(optional = true, mappedBy="documento")
    private Persona dueño;

    //...
}
```

Se omite para el caso de una navegabilidad unidireccional entre persona y documento.



*El ejemplo completo es para una navegabilidad bidireccional*





# JPA – Mapeos bidireccionales

- En el mapeo con navegabilidad bidireccional hay siempre un rol "**dueño**" de la relación.
- El **dueño** es quien configura las especificaciones físicas de la asociación, utilizando por ejemplo:
  - `@JoinColumn` para *claves foráneas*
  - `@JoinTable` para uso de *tablas de intersección*
- El rol "**no dueño**" de la asociación (el otro extremo) solamente utiliza "***mappedBy***" para indicar el nombre del atributo de la otra clase la cual posee la configuración de mapeo.
- El atributo ***mappedBy*** podrá ser utilizado en las relaciones `@OneToOne`, `@OneToMany` y `@ManyToMany`. Solamente `@ManyToOne` no acepta este atributo.



# JPA – One-To-Many y Many-To-One

@Entity

```
public class Continente {
```

```
//...
```

```
@OneToMany(mappedBy="continente")
```

```
private List<Pais> paises;
```

```
//..
```

```
}
```

@Entity

```
public class Pais {
```

```
//...
```

```
@ManyToOne(optional = false)
```

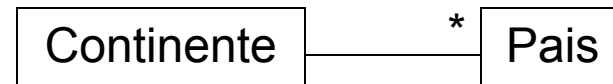
```
@JoinColumn(name="continente_id")
```

```
private Continente continente;
```

```
//...
```

```
}
```

Para las relaciones *One-To-Many* y *Many-To-Many* el tipo de las colecciones debe ser una interface: **Collection**, **List**, **Set** o **Map**



*El ejemplo completo es para una navegabilidad bidireccional, para una navegabilidad unidireccional es posible usar @JoinColumn con @OneToMany (JPA 2.0) o con @ManyToOne y omitir el mappedBy*



# JPA – Many-To-Many

**@Entity**

```
public class Alumno {
```

```
    @Id
```

```
    @Column(name="ALUMNO_ID")
```

```
    private int id;
```

```
    //...
```

```
    @ManyToMany
```

```
    @JoinTable(name="ALUMNO_CURSO",
```

```
        joinColumns=@JoinColumn(name="ALU_ID",
```

```
            referencedColumnName="ALUMNO_ID"),
```

```
        inverseJoinColumns=@JoinColumn(name="CUR_ID",
```

```
            referencedColumnName="CURSO_ID")
```

```
    )
```

```
    private List<Curso> cursos;
```

```
    //...
```

```
}
```

**@Entity**

```
public class Curso {
```

```
    @Id
```

```
    @Column(name="CURSO_ID")
```

```
    private int id;
```

```
    //...
```

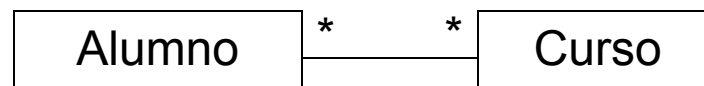
```
    @ManyToMany(mappedBy="cursos")
```

```
    private List<Alumno> alumnos;
```

```
    //...
```

```
}
```

**@JoinTable** no es necesaria, solo es para dejar prolija la relación entre *Alumno* y *Curso*



*El ejemplo completo es para una navegabilidad bidireccional, para una navegabilidad unidireccional solo incluir el atributo “dueño”*



# JPA – Lectura ansiosa y perezosa

En JPA, la lectura de las relaciones puede ser de dos tipos:

- **EAGER**: indica que la relación se recupera cuando se cargue el objeto
- **LAZY**: indica que la relación se recupera cuando se consulte la relación sobre el objeto

Por defecto el tipo de lectura para **One-To-One** y **Many-To-One** es *ansiosa* (**EAGER**).

Por defecto el tipo de lectura para **One-To-Many** y **Many-To-Many** es *perezosa* (**LAZY**).

Desventajas:

**LAZY**: tenemos que asegurarnos que la relación no se necesite después de que el objeto se haya desvinculado de la sesión (*LazyInitializationException*).

**EAGER**: puede dar como resultado una carga excesiva de datos, obteniendo problemas de performance y limitaciones de memoria

*Ejemplo:*

```
@ManyToOne(fetch=FetchType.LAZY)
```



# JPA - Transitividad de operaciones

La opción **cascade** que nos permite realizar *operaciones en cascada sobre las relaciones* y generalmente se utiliza en relaciones dependientes, de tipo padre-hijo (por ejemplo Factura-Detalle)

Existen **cinco opciones** para **cascade**:

- **PERSIST**: se aplica al llamar al método *persist()* del EntityManager.  
Si *persist()* se llama desde el padre y el hijo es nuevo, éste también se persistirá.  
Si existe el hijo, no pasará nada.  
Sin la opción PERSIST, se requerirá persistir primero el hijo y después el padre.
- **REMOVE**: se aplica al llamar al método *remove()* del EntityManager.  
Si *remove()* se llama desde el padre, todos sus hijos se eliminarán.  
Debería ser utilizado por relaciones dependientes.  
Sacar un objeto dependiente de la colección *OneToMany* no implica un borrado del objeto dependiente, es necesario efectuar un *remove()* sobre el mismo.



# JPA - Transitividad de operaciones

- **MERGE**: se aplica al llamar al método *merge()* del EntityManager.  
Si *merge()* se llama desde el padre, todos sus hijos serán merged.  
Debería ser utilizado por relaciones dependientes.
- **REFRESH**: se aplica al llamar al método *refresh()* del EntityManager.  
Si *refresh()* se llama desde el padre, los hijos también serán refrescados.  
Debería ser utilizado por relaciones dependientes.
- **ALL**: se aplican todas las opciones anteriores.

*Ejemplo:*

```
@OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
```



# JPA - Herencia entre entidades

En JPA se definen tres estrategias para mapear las relaciones de herencia:

- **SINGLE\_TABLE**: una sola tabla para guardar toda la jerarquía de clases.  
Ventaja: es la opción de **mejor rendimiento** porque accede a una tabla que está totalmente desnormalizada.  
Desventaja: todos los campos de las clases hijas tienen que admitir nulos  
Es la opción default de JPA
- **JOINED**: una tabla por cada clase. Se mapea cada clase a una tabla separada, unidas por FK.  
Ventaja: esquema normalizado por lo tanto **la mas flexible**  
Desventaja: cualquier consulta que no sea a la superclase requiere *joins*
- **TABLE\_PER\_CLASS**: una tabla por cada clase concreta, repitiendo los campos de la superclase en cada tabla de subclase  
El identificador es el mismo para toda la jerarquía: el de la superclase  
Si la superclase es concreta, se necesita otra tabla para ella  
Ventaja: únicamente en ausencia de comportamiento polimorfo  
Desventaja: hacer consultas buscando objetos que podrían estar en cualquiera de las subclases requiere múltiples queries o uniones  
En la versión 3.0 de EJBs, está especificada pero no es obligatoria su implementación



# JPA - Herencia entre entidades

*Ejemplo de de estrategia SINGLE\_TABLE*

**@Entity**

**@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)** // o **@Inheritance**

**@DiscriminatorColumn(name="TIPO\_VEHICULO")**

**public abstract class Vehiculo {**

**@Id**

**private long id;**

**...**

**}**

*..el tipo de datos en el discriminador  
puede ser CHAR, STRING, INTEGER*

**@Entity**

**@DiscriminatorValue("A")**

**public class Auto extends Vehiculo {**

**...**

**}**

**@Entity**

**@DiscriminatorValue("C")**

**public class Camioneta extends Vehiculo {**

**...**

**}**





# JPA - Herencia entre entidades

*Ejemplo de de estrategia JOINED*

**@Entity**

**@Inheritance(strategy = InheritanceType.JOINED)**

**public abstract class Vehiculo {**

**@Id**

**private long id;**

**...**

**}**

**@Entity**

**public class Auto extends Vehiculo {**

**...**

**}**

**@Entity**

*//especificación física opcional*

**@PrimaryKeyJoinColumn(name="CAMIO\_ID" referencedColumnName="ID")**

**public class Camioneta extends Vehiculo {**

**...**

**}**