JAVA PERSISTENCE API (JPA)

Ing. Ronald Cuello Meza

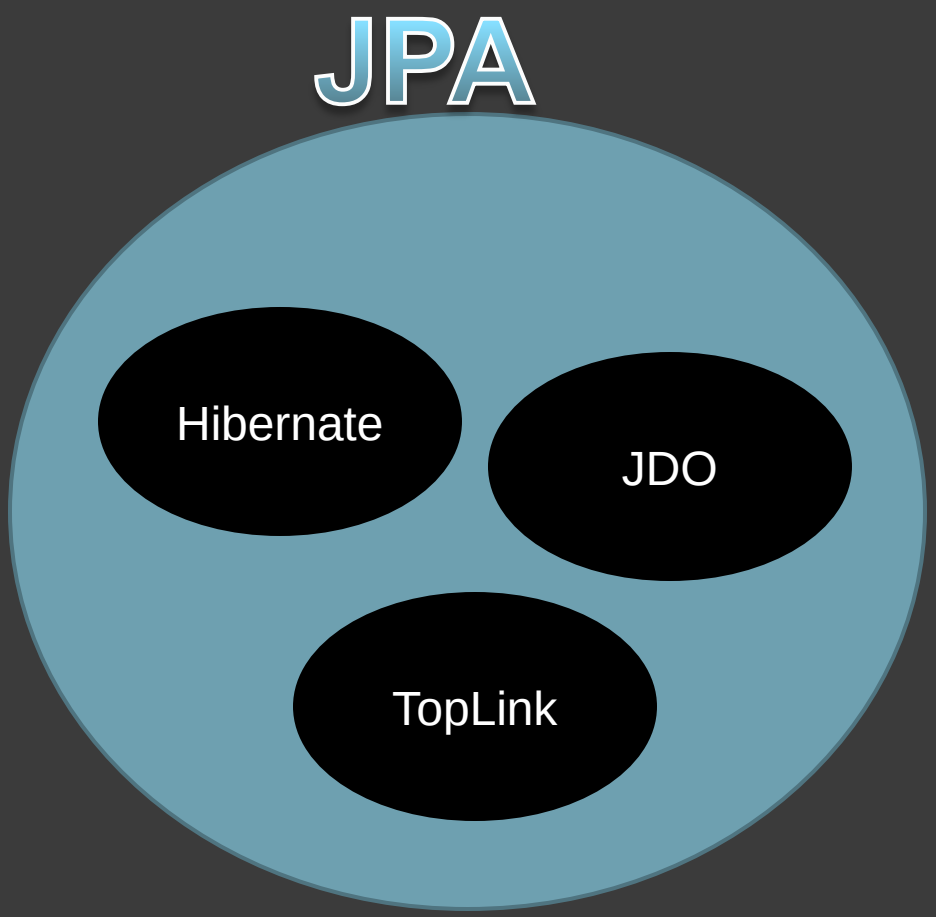
JPA: Java Persistence API

- **Java Persistence API** (JPA) proporciona un **estándar** para gestionar datos relacionales en aplicaciones Java SE o Java EE, de forma que además se **simplifique el desarrollo** de la persistencia de datos.

  Es una API de persistencia de POJOs (Plain Old Java Object). Es decir, objetos simples que no heredan ni implementan otras clases (como los EJBs).

  Requerimientos:
  - Programación Orientada a objetos
  - Anotaciones en Java 5
  - Generic en java 5
  - Base de datos relacionales
  - Lenguaje estructurado de consultas (SQL)

JPA: Java Persistence API



En su definición, ha combinado ideas y conceptos de los principales frameworks de persistencia, como Hibernate, Toplink y JDO, y de las versiones anteriores de EJB. Todos estos cuentan actualmente con una implementación JPA.

El mapeo objeto-relacional (es decir, la relación entre entidades Java y tablas de la base de datos, *queries* con nombre, etc) se realiza mediante anotaciones en las propias clases de entidad. No se requieren ficheros descriptores XML. También pueden definirse transacciones como anotaciones JPA.

JPA: Java Persistence API

El mapeo objeto/relacional, es decir, la relación entre entidades Java y tablas de la base de datos, se realiza mediante anotaciones en las propias clases de entidad, por lo que no se requieren ficheros descriptores XML. También pueden definirse transacciones como anotaciones JPA.

JPA: Java Persistence API

Las anotaciones de JPA se clasifican en dos categorías:

Permiten describir modelo de objeto, asociaciones de clase, etc

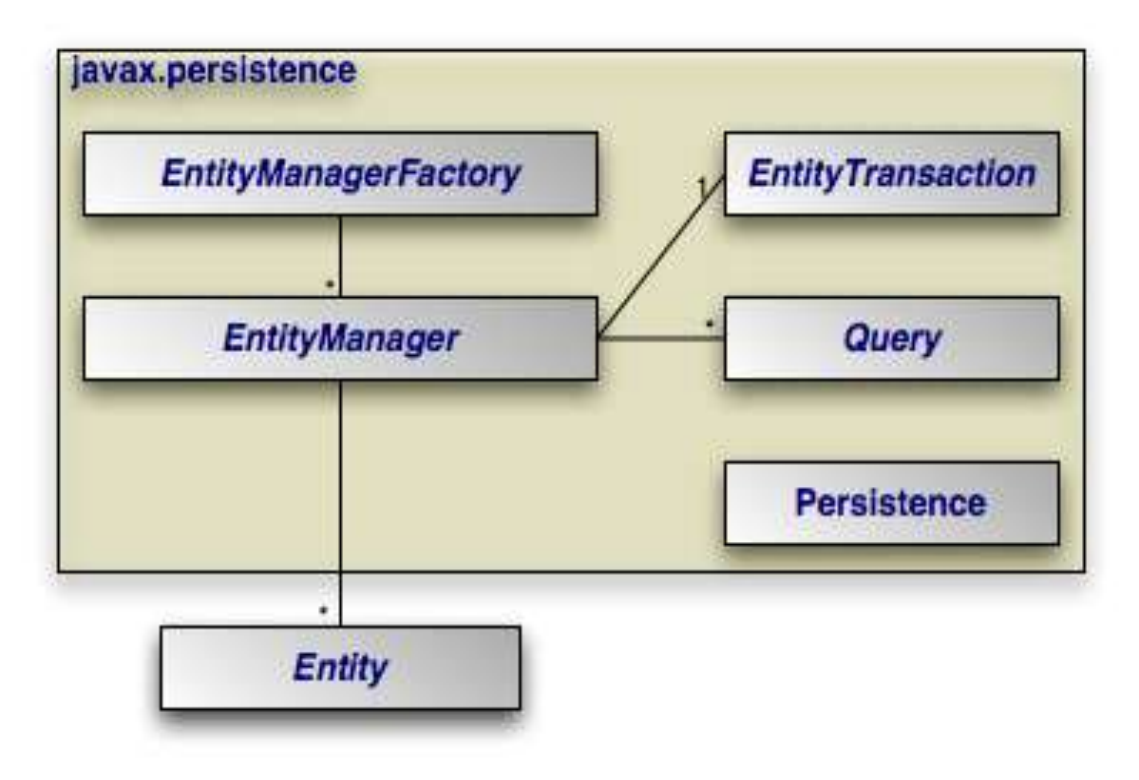
@OneToOne,@OneToMany,etc

Describen esquemas físicos de base de datos, tablas, columnas, índices, etc.

@Column,@JoinColumn,etc

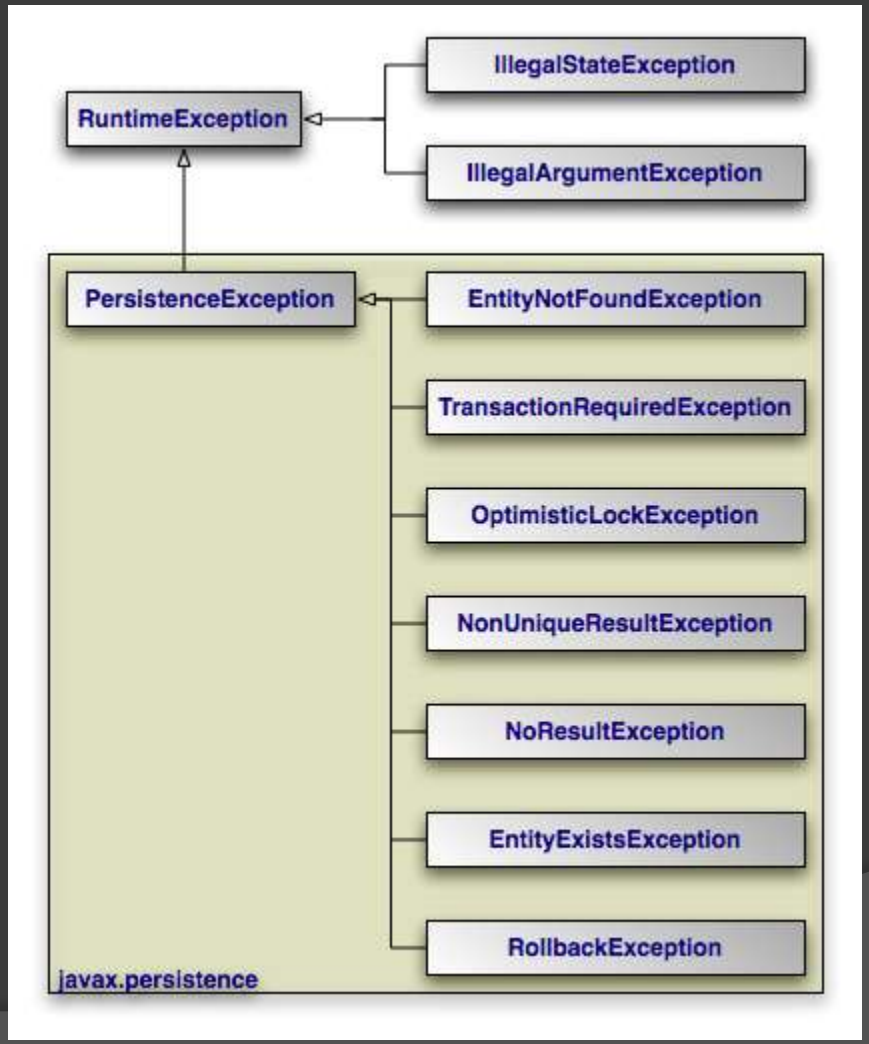
Arquitectura

El siguiente diagrama muestra la relación entre los componentes principales de la arquitectura de JPA:

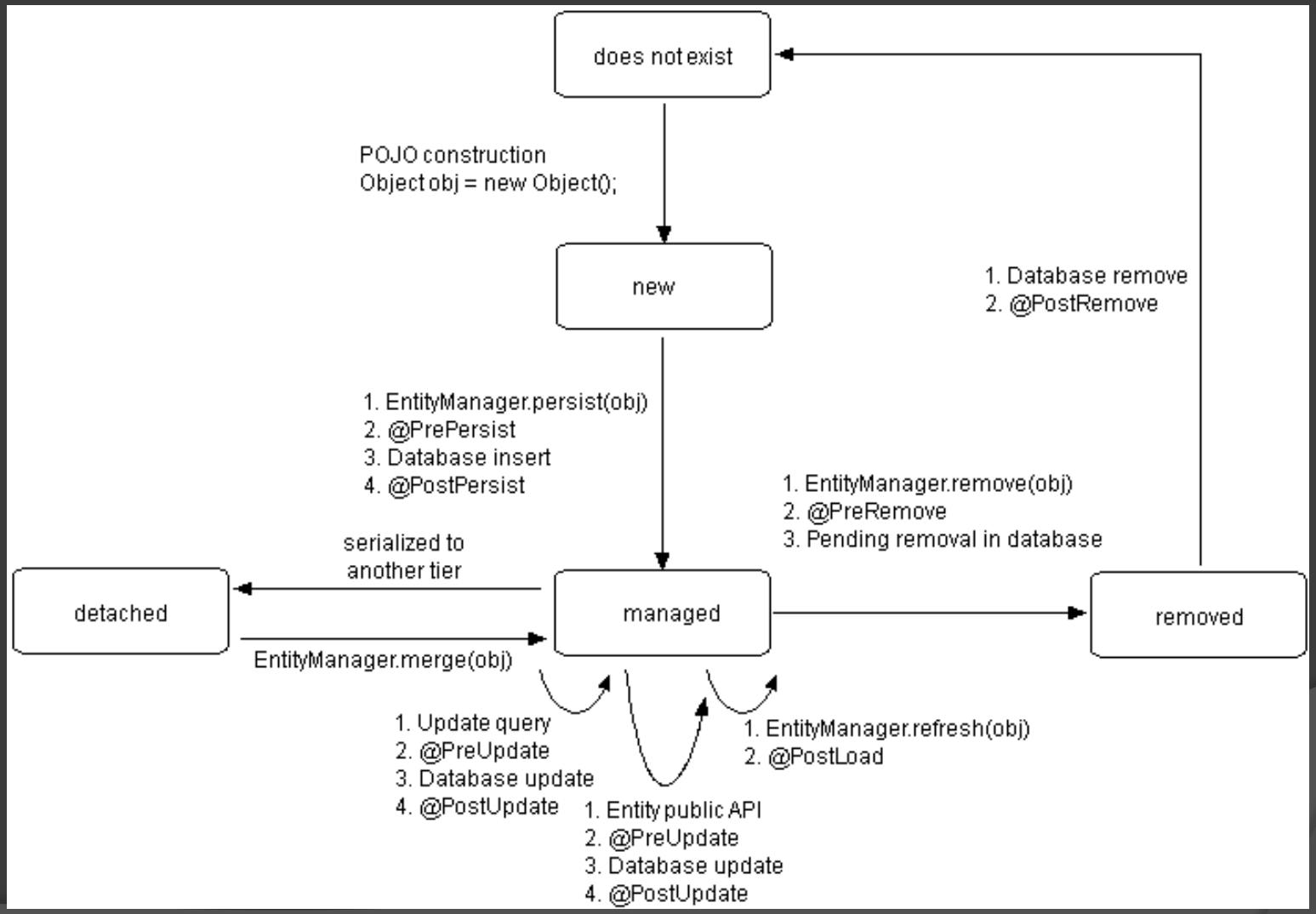


Excepciones

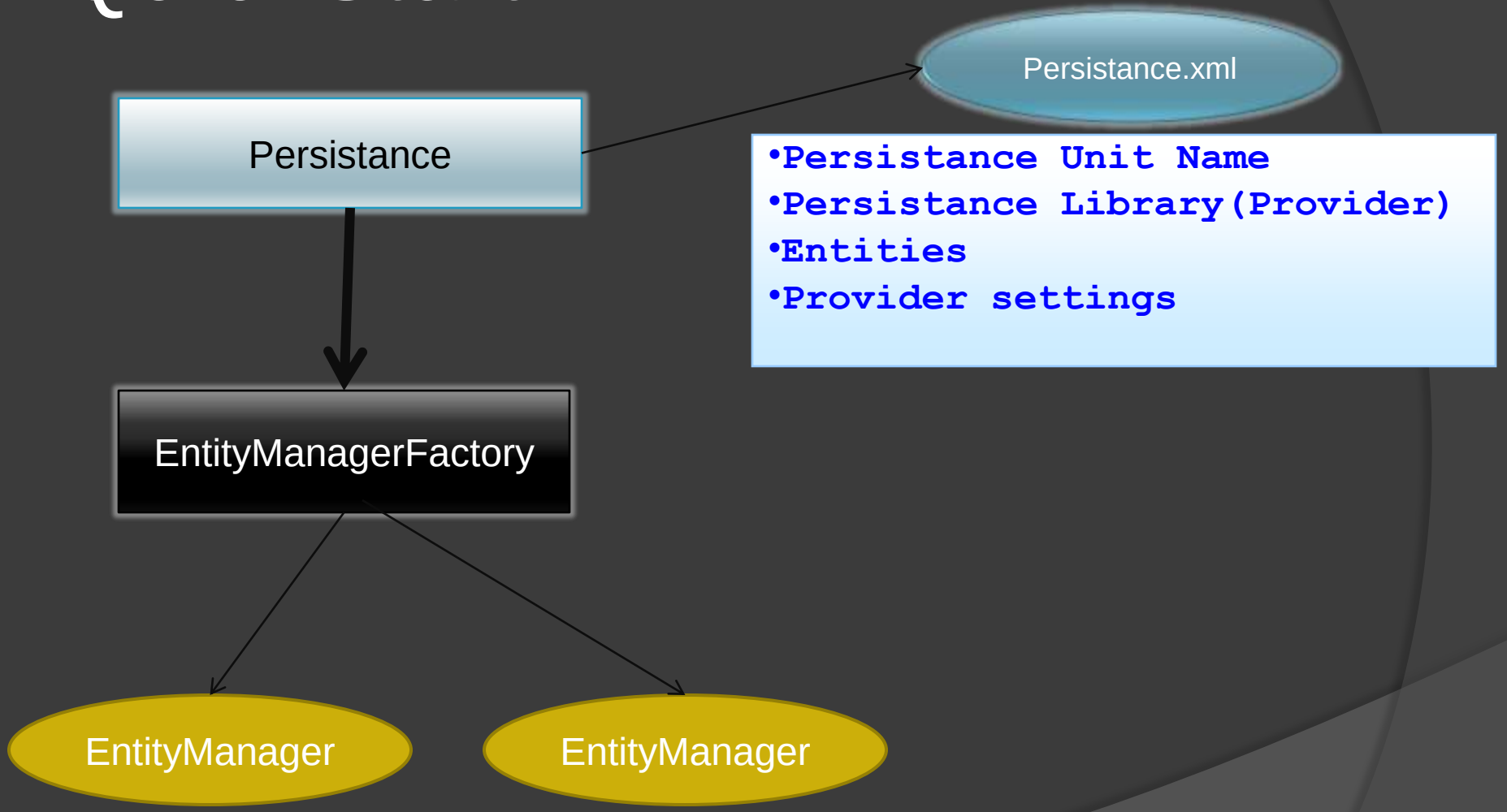
A continuación mostraremos la arquitectura que JPA tiene para el control de las excepciones:



Ciclo de Vida de una Entidad



QuickStart


Persistance.xml
Persistance
•Persistance Unit Name
•Persistance Library(Provider)
•Entities
•Provider settings
EntityManagerFactory
EntityManager
EntityManager

Operaciones con Entidades

Las siguientes son operaciones que pueden ser ejecutadas en objetos de entidad(Entity) con la ayuda de la API EntityManager.
EntityManager son objetos que nos ayudan a administrar uno o mas entidades con la ayuda de un Contexto de persistencia implicito..

Persistiendo una Entidad

Operaciones con Entidades

EntityManager.persist(Entidad).
Entidades son simples pojos (Plain old Java objets) hasta que entran en estado managed(manejado) y luego hechos persistentes por un EntityManager.

La siguiente linea de codigo nos muestra una entidad persistente y manejada.

```
Persona persona=new Persona();
persona.set(...)
entityManager.persist(persona);
```

Cuando el metodo persist() es llamado,el motor de persistencia revisará por la existencia del objeto con la ayuda de su identificador unico(usualmente representada en la forma de llave primaria).Si un objeto duplicado es encontrado,entonces se lanzará una RunTime exception, **EntityExistsException.**

Buscando Entidades

Operaciones con Entidades

EntityManager.find(class , object)

El método find() recibe como parámetros la clase de la entidad y valor de la llave primaria para buscar un solo objeto entidad.

Si el objeto que estamos buscando no puede ser localizado por la clase EntityManager,entonces este método simplemente retornará null.Ver la siguiente línea de código:

```
Long pk=1;
Persona persona=EntityManager.find(Persona.class,pk);
If(persona!=null){
   //Hacer algo con la persona...
}
```

lo bueno sobre el método find() es que la entidad se convierte en objeto manejado en nuestro contexto de persistencia del EntityManager.(ver diagrama del ciclo de vida de la entidad)

Operaciones con Entidades

EntityManager.getReference(class , object)

Este método es similar al método find(),toma como argumentos el nombre de la clase y el valor de la llave primaria.
Pero a diferencia del método find() ,que retorna null sin no encuentra la entidad ,este método nos lanzará una ***EntityNotFoundException.***

Otra diferencia es que la entidad extraída por este método ,algunos datos de la entidad pueden presentar carga perezosa (lazy loading)al ser accedidos la primera vez.

```
Long pk=1;
Try{
Persona persona=EntityManager.getReference(Persona.class,pk);
}
catch(EntityNotFoundException ex)
{
...
}
```

Eliminando Entidades

Operaciones con Entidades

EntityManager.remove(entidad)

Para eliminar una entidad de la base de datos se debe invocar el metodo ***EntityManager.remove(EntityObject).***
Para poder eliminar una entidad ,esta debe ser una entidad manejada por que sino la operación fallará.

Luego de invocar este metodo,la entidad entrará a fase desprendido (detached) del contexto de persistencia y no será una entidad manejada.

```
entityManager.getTransaction().begin();

Persona persona=entityManager.find(Persona.class,1L);
entityManager.remove(persona);

entityManager.getTransaction().commit();
```

Actualizando Entidades

Operaciones con Entidades

EntityManager.merge(entidad)

El método EntityManager.merge(entidad) lo que hace es tomar una entidad en estado detached y asociarla al contexto de persistencia del entityManager,actualizando los datos del objeto en la base de datos. Considerando la siguiente línea de código:

La entidad entra en estado manejado

```
//Transaction Begin
Persona persona=entityManager.find(Persona.class,1L);
//Transaction Ends
persona.setNombre("nuevo nombre");
entityManager.merge(persona);//otra instancia de EntityMan
```

Transacción completa, contexto de persistencia se apaga, la entidad pasa a estado detached. Toda modificación que hagamos si el contexto de persistencia está apagado, no realizará ninguna operación frente a la base de datos debido a que el objeto se encuentra en estado detached.

Ahora ,invocando el método merge(),tomará un objeto y lo pasará a estado manejado ,y todos los cambios realizados en este objeto serán visibles al contexto de persistencia del EntityManager.

Operaciones con Entidades

EntityManager.merge(entidad)

Tenemos el siguiente ejemplo:

DAOPersona.java

```
Persona public static Persona find(Long id){
        EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
        Persona persona=null;
        try{
            persona=em.find(Persona.class, id);
        }catch(Exception ex){
            System.out.println("upss!! ha ocurrido un error");
            ex.printStackTrace();
        }
        finally{
            em.close();
        }
        return persona;
    }
```

Nota : la clase JpaUtil es un inicializador de nuestro motor de persistencia

Operaciones con Entidades

DAOPersona.java

```
public static void update(Persona persona){
        EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
        EntityTransaction tx=em.getTransaction();
        tx.begin();
        try{
            em.merge(persona);
            tx.commit();
            System.out.println("Actualizacion exitosa");
        }catch(Exception ex){
            tx.rollback();
            System.out.println("Ha ocurrido un erro al actualizar");
            ex.printStackTrace();
        }
        finally{
            em.close();
        }
    }
```

Operaciones con Entidades

Main.java

```
public static void main(String[] args){
        Persona p=DAOPersona.find(3L);

        System.out.println("-----valores de base de datos------");
        System.out.println(p);
        System.out.println("------Cambios---------");

        p.setNombre("Ronald");
        p.setApellido("Cuello");

        DAOPersona.update(p);
}
```

Flush y Refresh

Operaciones con Entidades

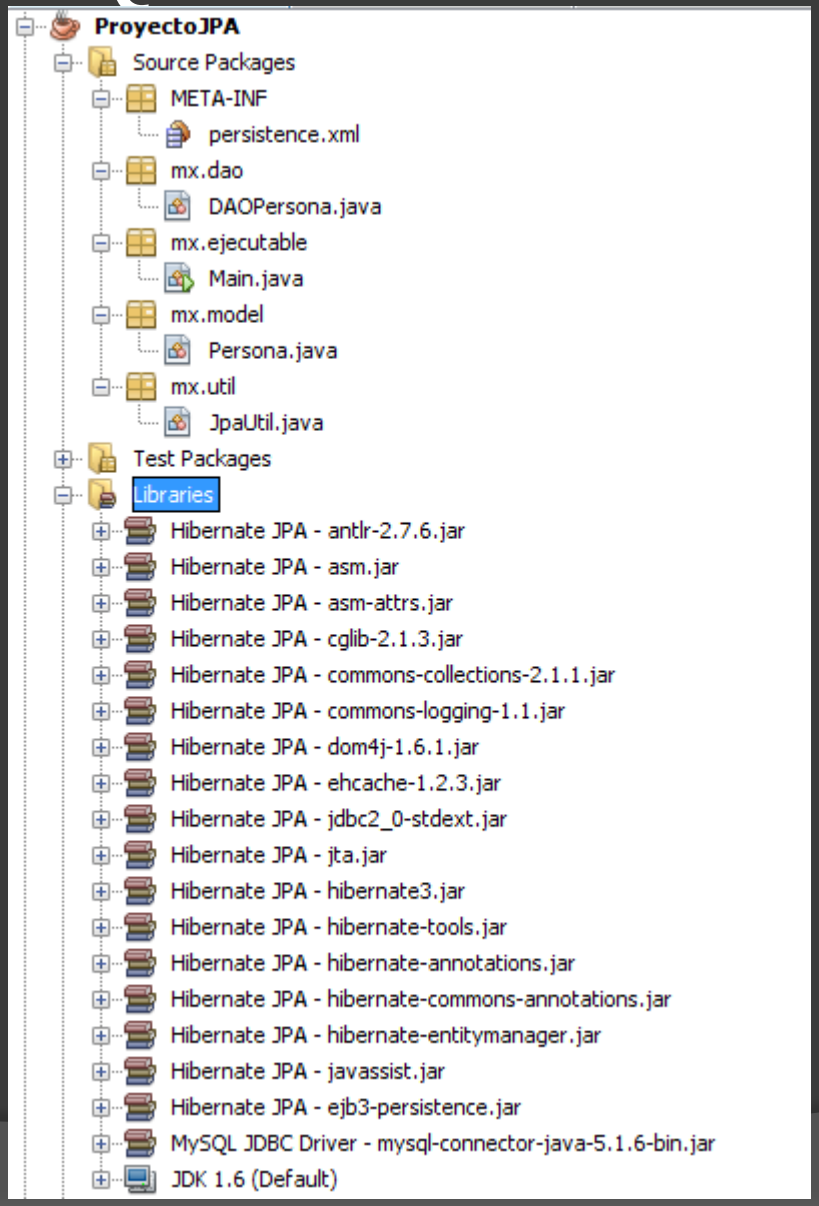
EntityManager.flush()

Sincroniza los datos de una entidad y los hace persistentes en la base de datos.

EntityManager.refresh(entidad)

Contrario al flush() ,este método vuelve a cargar los datos originales de la base de datos a la entidad.

QuickStart

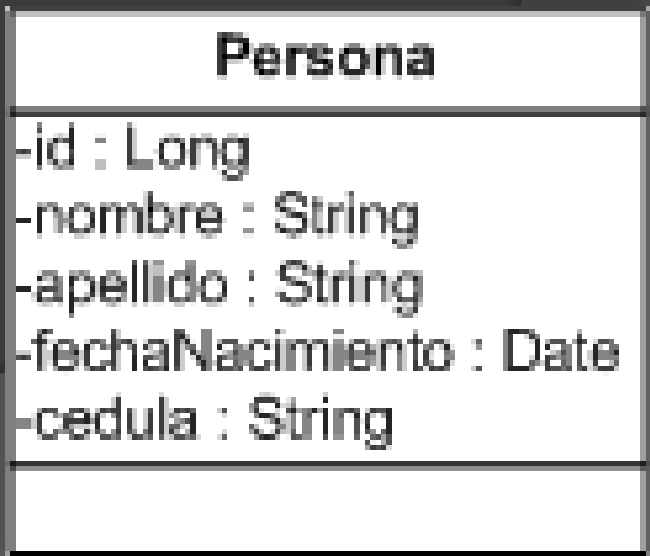


Estructura de proyecto de ejemplo

Tabla en base de datos

personas	
PK	id
	nombre apellido fecha_nacimiento cedula

Clase Persona.java



QuickStart

Creacion de la clase Persona.java

```
Package mx.model;
public  class Persona implements Serializable{
    @Id
    @Column(name="id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private  Long id=null;
    @Column(name="nombre")
    private String nombre;
    @Column(name="apellido")
    private String apellido;
    @Column(name="fecha_nacimiento")
    @Temporal(TemporalType.DATE)
    private Date fechaNacimiento;
    @Column(name="cedula")
    private String cedula;
    //setters and getters
}
```

QuickStart

Archivo de configuración del motor de persistencia persistence,xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" 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_1_0.xsd">
  <persistence-unit name="ProyectoJPA_PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>mx.model.Persona</class>
<properties>
      <property name="hibernate.connection.username" value="username"/>
      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.password" value="password"/>
      <property name="hibernate.connection.url"
value="jdbc:mysql://localhost:3306/mi_base_de_datos"/>
      <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider"/>
    </properties>
  </persistence-unit>
</persistence>
```

QuickStart

Creacion de la clase JpaUtil.java

```
Package mx.util;
public  class JpaUtil {
    private static final EntityManagerFactory emf;
    static{
        try{
            emf=Persistence.createEntityManagerFactory("ProyectoJPA_PU");
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static EntityManagerFactory getEntityManagerFactory(){
        return emf;
    }
}
```

QuickStart

Creación de la clase Persona.java

```
@Entity
@Table(name="personas")
public class Persona implements Serializable{
    private static final long serialVersionUID = -4365400650816983 26L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private  Long id=null;

    @Column(name="nombre")
    private String nombre;

    @Column(name="apellido")
    private String apellido;

    @Column(name="cedula")
    private String cedula;

    @Temporal(TemporalType.DATE)
    @Column(name="fecha_nacimiento")
    private Date fechaNacimiento;
    //setters and getters
```

QuickStart

Creación de la clase DAOPersona.java

```
public static Persona find(Long id){
        EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
        Persona persona=null;
        try{
            persona=em.find(Persona.class, id);
        }catch(Exception ex){
            System.out.println("upss!! ha ocurrido un error");
            ex.printStackTrace();
        }
        finally{
            em.close();
        }
        return persona;
    }
```

QuickStart

Creación de la clase DAOPersona.java

```
public static void create(Persona persona){
        EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
        em.getTransaction().begin();
        try{
            em.persist(persona);
            em.getTransaction().commit();

        }catch(Exception ex){
            em.getTransaction().rollback();
            System.out.println("Ha ocurrido un error al guardar");

        }
        finally{
            em.close();
        }
    }
```

QuickStart

Creación de la clase DAOPersona.java

```
public static void update(Persona persona){
        EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
        EntityTransaction tx=em.getTransaction();
        tx.begin();
        try{
            em.merge(persona);
            tx.commit();
            System.out.println("Actualizacion exitosa");
        }catch(Exception ex){
            tx.rollback();
            System.out.println("Ha ocurrido un erro al actualizar");
            ex.printStackTrace();
        }
        finally{
            em.close();
        }
    }
```

QuickStart

Creación de la clase DAOPersona.java

```
public static void delete(Long id){
        EntityManager em=JpaUtil.getEntityManagerFactory().createEntityManager();
        em.getTransaction().begin();
        try{


            Persona p=DAOPersona.find(id);
            em.remove(p);
            em.getTransaction().commit();

        }catch(Exception ex){
            ex.printStackTrace();
        }
        finally{
            em.close();
        }
    }
```

Entidad (Entity)

Una entidad es un objeto de dominio de persistencia. Normalmente, una entidad representa una tabla en el modelo de datos relacional y cada instancia de esta entidad corresponde a un registro en esa tabla.

El estado de persistencia de una entidad se representa a través de campos persistentes o propiedades persistentes. Estos campos o propiedades usan anotaciones para el mapeo de estos objetos en el modelo de base de datos.

```
@Entity
public class Empleado implements Serializable {
}
```

Entidad (Entity)

Para definir los datos de una tabla de base de datos, utilizamos la anotación @**Table** a nivel de la declaración de la clase, esta anotación nos permite definir el nombre de tabla con la que se está mapeando la clase, el esquema, el catálogo, constraints de unicidad. Si no se utiliza esta anotación y se utiliza únicamente la anotación Entity, el nombre de la tabla corresponderá al nombre de la clase.

```
@Entity
@Table(name="empleados")
public class Empleado implements Serializable {
}
```

La anotación **Table** contiene los atributos de catalog y schema, en caso de que necesitemos definirlos. Usted también puede definir constraints de unicidad utilizando la anotación @UniqueConstraint en conjunción con la anotación @Table

```
@Table(name="empleados",uniqueConstraints=
{
   @UniqueConstraint(columnNames={"salario","fecha_ingreso"}
})
```

Entidad (Entity)

Requerimientos para una Clase Entidad

La clase debe poseer en su encabezado la anotación *javax.persistence.Entity @Entity*

Ningún método o variables de instancias deben ser declaradas como FINAL

```
@Entity
public class Empleado implements Serializable {
        private Long id;
        //setId getId
        public Empleado(){
           ...
        }
}
```

Los atributos persistentes de la clase deben ser declarados como PRIVATE,PROTECTED o package-private,y solo deben ser accesadas por los metodos de la clase

Si una instancia de una entidad entra en entornos remotos ,debe implementar la interfaz SERIALIZABLE

La clase debe poseer un constructor PUBLIC,PROTECTED sin argumentos

La clase no debe ser declarada como FINAL

Entidad (Entity)

El estado persistente de una entidad puede ser accesible a través de variables de instancia a la entidad o bien a través de las propiedades de estilo de JavaBean(Setters and Getters). Los campos o propiedades pueden tener asociados los siguientes tipos Java:

- Tipos primitivos de Java (int,long,double,etc)
- java.lang.String
- Otro tipo de objeto serializable, incluyendo:
  - Wrappers de tipos primitivos en Java (Integer,Long,Double,etc)
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - User-defined serializable types
  - byte [ ]
  - Byte [ ]
  - char [ ]
  - Character [ ]
  - Tipos enumerados (Enumeration)
  - Otras entidades y/o colecciones de entidades

Entidad (Entity)

Las entidades podrán utilizar **campos persistentes o propiedades persistentes**.

Si las anotaciones de mapeo se aplican a las instancias de las entidades, la entidad utiliza campos persistentes, En cambio, si se aplican a los métodos getters de la entidad, se utilizarán propiedades persistentes. Hay que tener en cuenta que no es posible aplicar anotaciones tanto a campos como a propiedades en una misma entidad.

1 **Campos persistentes**

Si la entidad utiliza campos persistentes, los accesos se realizan en tiempo de ejecución. Aquellos campos que no tienen anotaciones del tipo javax.persistence.Transient o no han sido marcados como Java transitorio serán persistentes para el almacenamiento de datos. Las anotaciones de mapeo objeto/relación deben aplicarse a los atributos de la instancia.

Entidad (Entity)

2 **Propiedades persistentes**

Si la entidad utiliza propiedades persistentes, la entidad debe seguir el método de los convenios de componentes JavaBeans.

Las propiedades de JavaBean usan métodos getters y setters en cuyo nombre va incluido el atributo de la clase al cual hacen referencia.

Si el atributo es booleano podrá utilizarse isProperty en lugar de getProperty.

Por ejemplo, si una entidad Empleado, utiliza las propiedades de persistencia, supongamos que tiene un atributo privado denominado nombre, la clase definirá los métodos getNombre y setNombre para recuperar y establecer el valor de la variable nombre.

Las anotaciones del mapeo objeto/relacional deben aplicarse a los métodos getter. El mapeo de las anotaciones no puede aplicarse a los campos o propiedades anotadas como @Transient o marcadas como transient.

Entidad (Entity)

1. **Campos Persistentes**

```
@Entity
public class Empleado implements Serializable {
        @Id @GeneratedValue(strategy=GenerationType.AUTO)
        @Column(name="id")
        private Long id;
        @Column(name="nombre")
        private String nombre;
}
```

2. **Propiedades Persistentes**

```
@Entity
public class Empleado implements Serializable {
        @Id @GeneratedValue(strategy=GenerationType.AUTO)
        @Column(name="id")
        public Long getId(){return this.id;}
}
```

Entidad (Entity)

Las colecciones posibles son:

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

Las variables genéricas de estos tipos también pueden ser utilizadas

```
public Set<Cargo> getCargos(){}
public void setCargos(Set<Cargo>){}
```

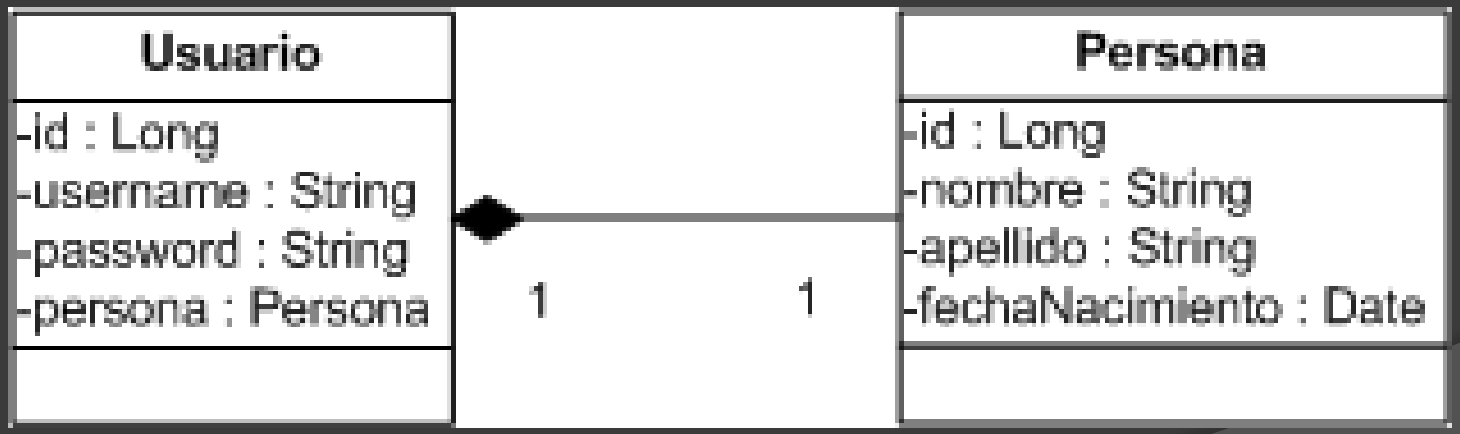
Asociaciones

Hay cuatro tipo de relaciones: uno a uno, uno a muchos, muchos a uno, y muchos a muchos.

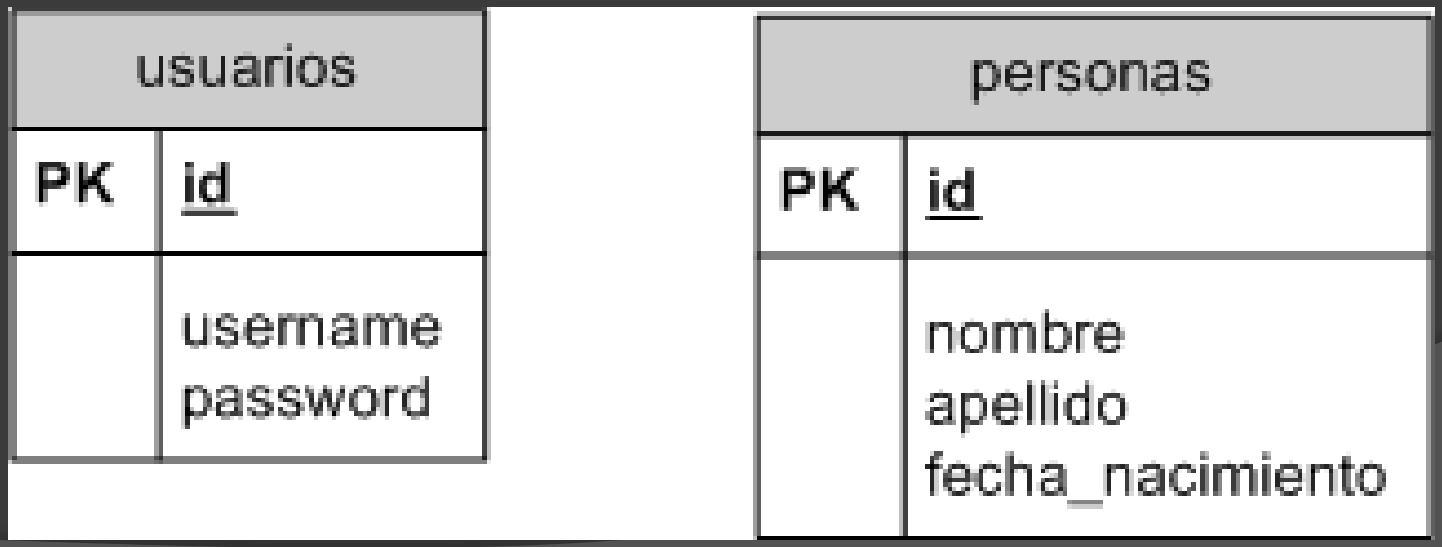
1

@OneToOne

Cada entidad se relaciona con una sola instancia de otra entidad. Donde la ambas se referencian por la misma llave PK.



Asociaciones



Asociaciones

@one-to-one

```
@Entity
@Table(name="usuarios")
public class Usuario implements Serializable {
      …//username password id
      @OneToOne
      @JoinColumn(name="id")
      private Persona persona;
      //setter and getter
}
```

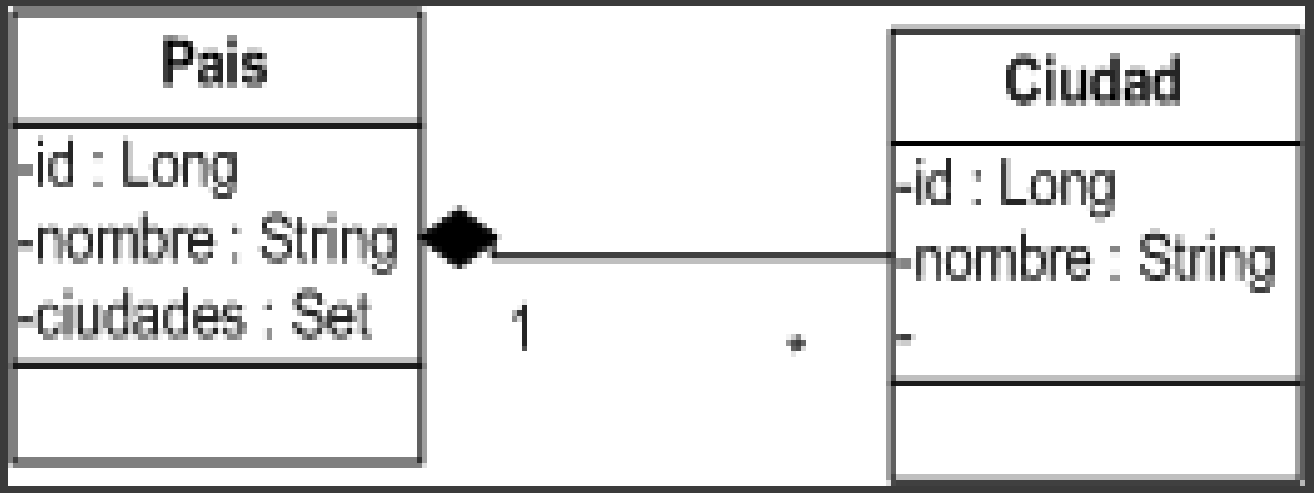
```
@Entity
@Table(name="personas")
public class Persona implements Serializable {
      …//id nombre apellido fechaNacimiento
}
```

Asociaciones

2

@OneToMany

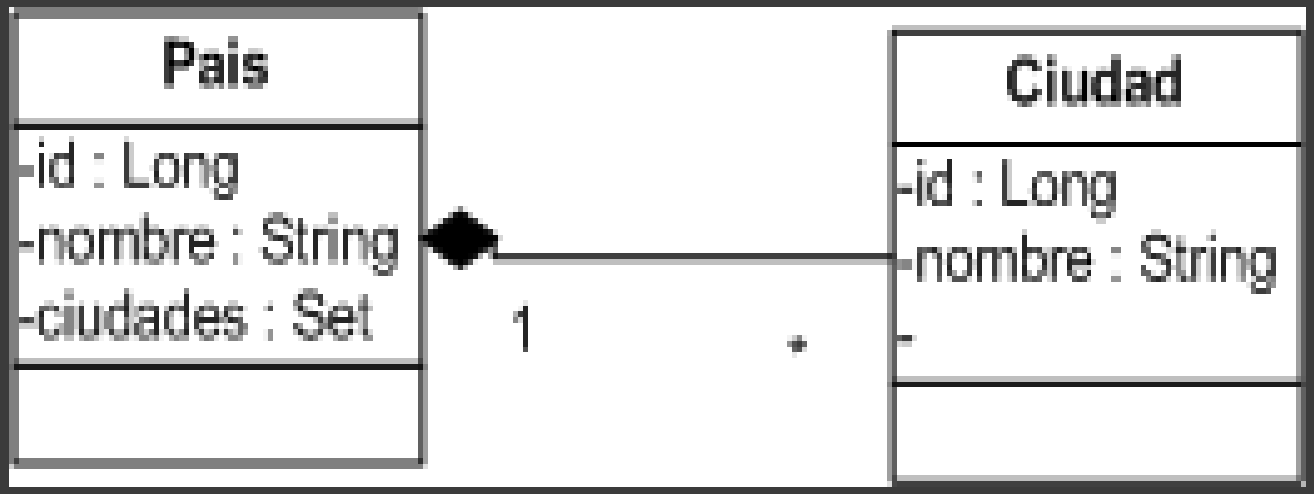
Una entidad, puede estar relacionada con varias instancias de otras entidades

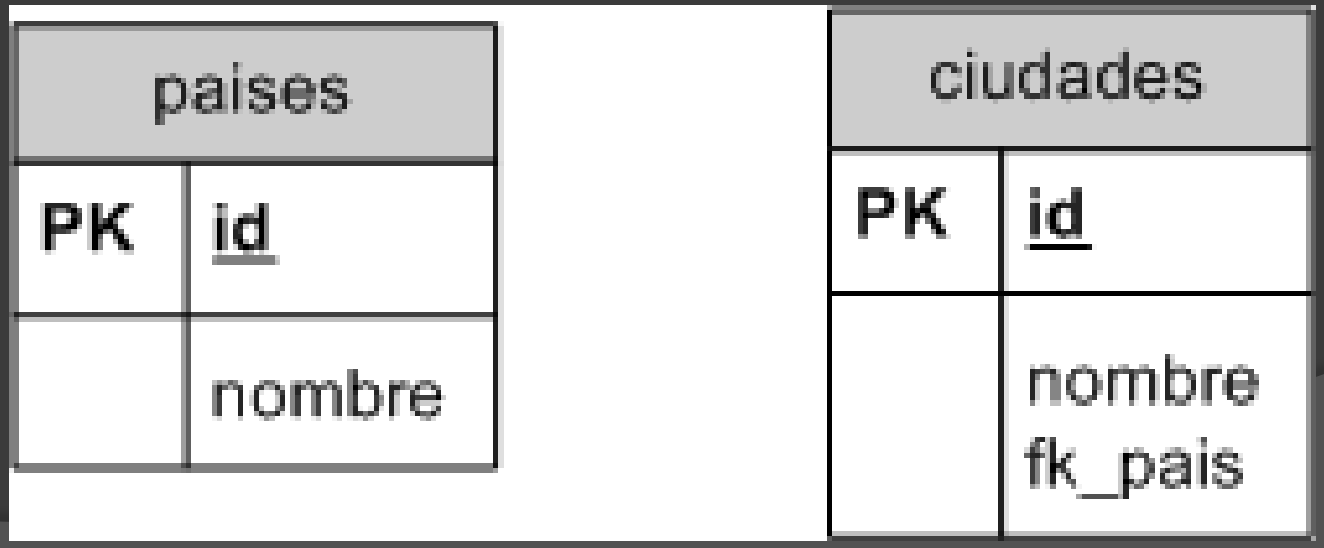



```
Clase Padre
@OneToMany(mappedBy="atributo_clase_hija",targetEntity=class)
```

```
Clase Hija
@ManyToOne
@JoinColumn(name="columna_referencia_tabla_padre")
```

Asociaciones





Asociaciones

@one-to-many

```
@Entity
@Table(name="paises")
public class Pais implements Serializable {
      …
    @OneToMany(mappedBy="pais",targetEntity=Ciudad.class)
    private Set<Ciudad>ciudades;
}
```

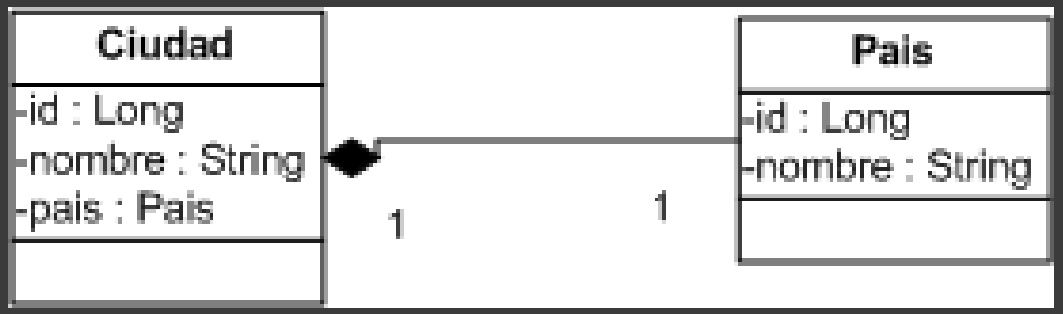
```
@Entity
@Table(name="ciudades")
public class Ciudad {
    …
    @ManyToOne
    @JoinColumn(name="fk_pais")
    private Pais pais;
}
```

Asociaciones

3

@ManyToOne

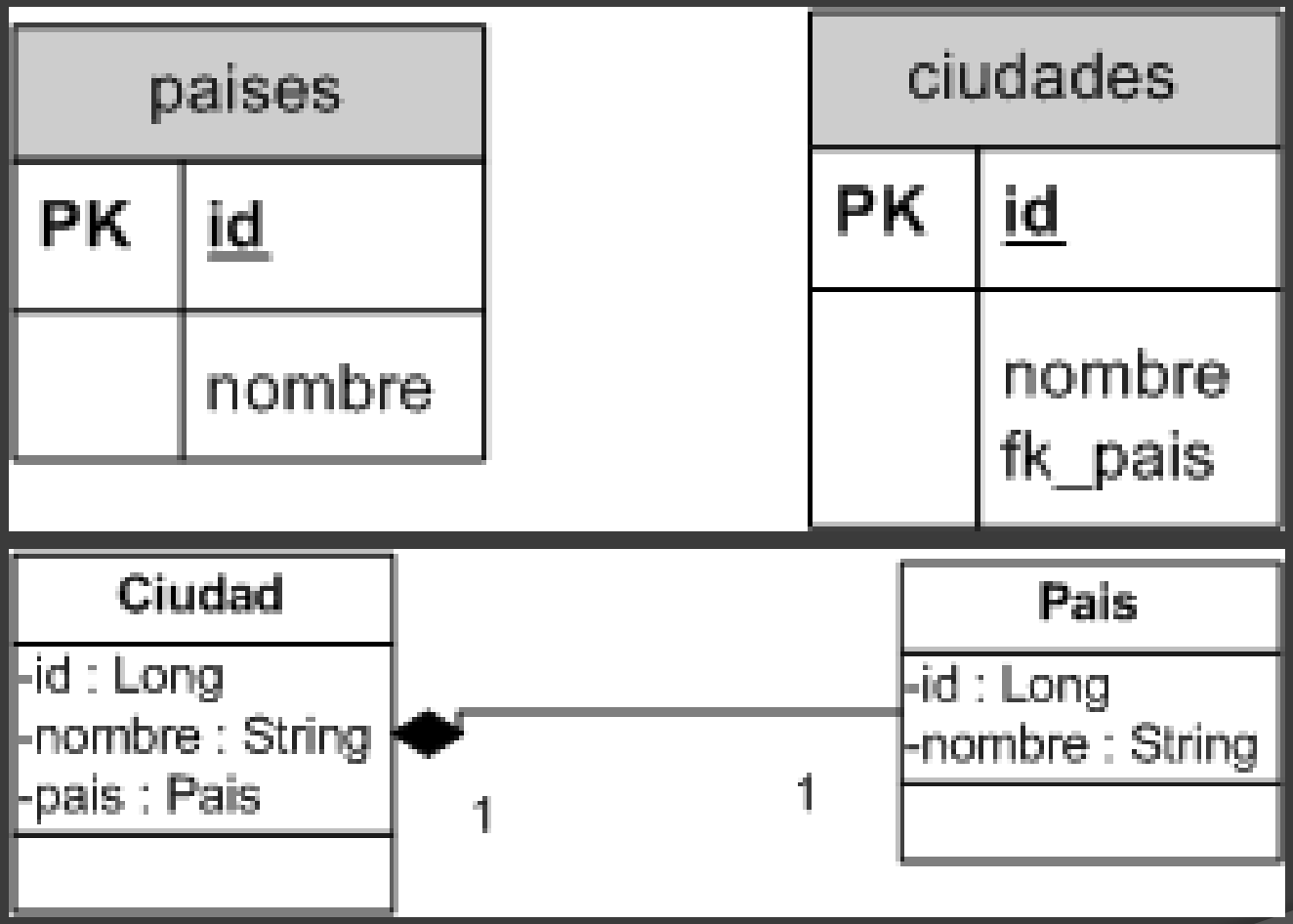
Múltiples instancias de una entidad pueden estar relacionadas con una sola instancia de otra entidad. Esta multiplicidad es lo contrario a la relación uno a muchos. Usado también como si fuera una relación OneToOne



Una ciudad solo puede estar en un pais,pero un país puede tener varias ciudades

```
Clase Hija
@ManyToOne
@JoinColumn(name="columna_referencia_tabla_padre")
```

Asociaciones



Asociaciones

@Many-to-One

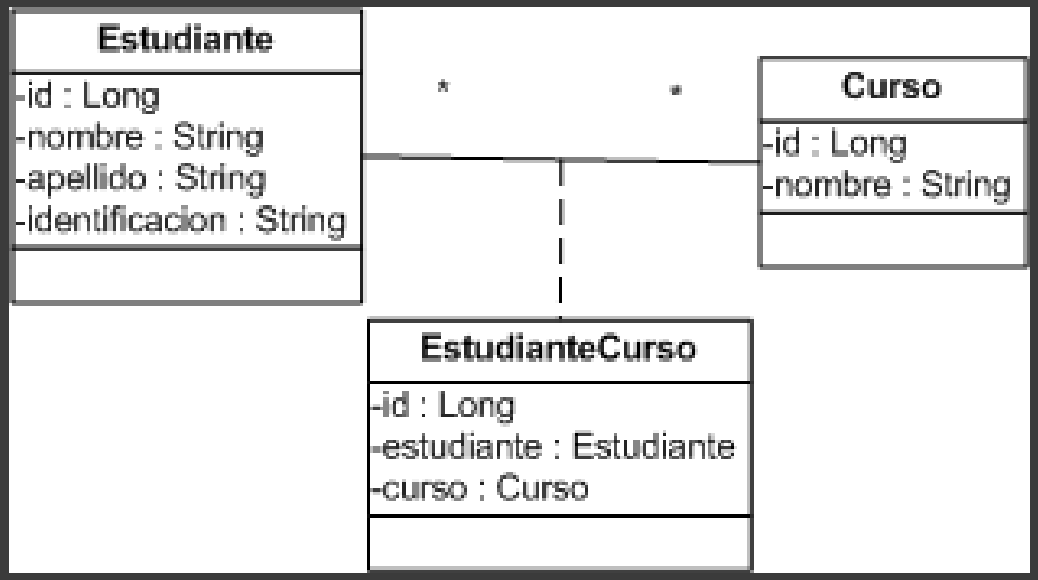
```
@Entity
@Table(name="ciudades")
public class Ciudad implements Serializable {
      …
    @ManyToOne
    @JoinColumn(name="fk_pais")
    private Pais pais;
}
```

Asociaciones

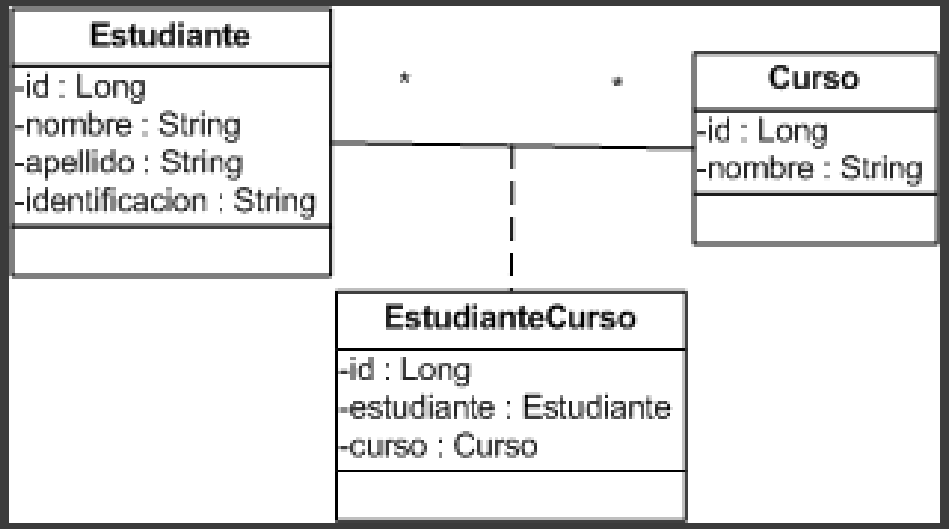
4

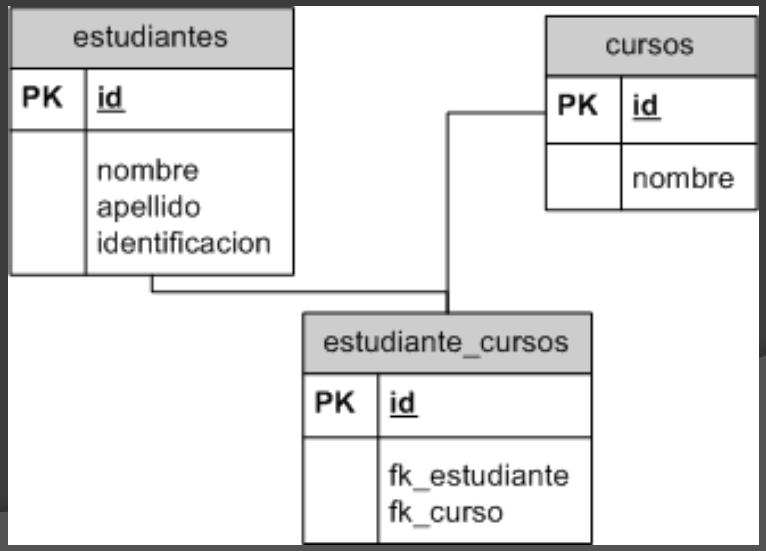
@ManyToMany

En este caso varias instancias de una entidad pueden relacionarse con múltiples instancias de otras entidades.



Asociaciones





Asociaciones

@Many-to-Many

```
@Entity
@Table(name="estudiantes")
public class Estudiante implements Serializable {
      …
    @ManyToMany
    @JoinTable(name="estudiante_cursos",
          joinColumns=@JoinColumn(name="fk_estudiante"),
          inverseJoinColumns=@JoinColumn(name="fk_curso"))
    private Set<Curso>cursos;
}
```

Asociaciones

@Many-to-Many

```
@Entity
@Table(name="cursos")
public class Curso implements Serializable{

    @Id
    @Column(name="id")
    private Long id;
    @Column(name="nombre")
    private String nombre;
    @Column(name="creditos")
    private Long creditos;
}
```

Java Persistence Query Language (JPQL)

Palabras Reservadas				
Select	From	Where	Update	Delete
Join	Outer	Inner	Left	Group
By	Having	Fetch	Distinct	Object
Null	True	False	Not	And
Or	Between	Like	In	As
Unknowm	Empty	Member	Of	Is
Avg	Max	Min	Sum	Count
Order	By	Asc	Desc	Mod
Upper	Lower	Trim	Position	Character_Length
Bit_Length	Current_Time	Current_Timestamp	New	Exists
All	Any	Some		

Java Persistence Query Language (JPQL)

Clausulas :

FROM

Existen varias formas de simplificar una clase a la hora de realizar la consulta

hql> FROM mx.model.Empleado

hql> FROM Empleado

Dando como resultado una colección de objetos de tipo Empleado

SELECT

la clausula SELECT selecciona cual(es) objeto (s)y cual (es) propiedad(es) se retornara en el resultado del query.

hql> SELECT v.marca FROM Vehiculo as v

hql> SELECT v.marca FROM Vehiculo v WHERE v.marca like ‘toy%’

hql> SELECT user.persona.nombre FROM Usuario user

hql> SELECT user.username,p.nombre FROM Usuario user,Persona p

La anterior consulta nos retorna Object[ ]

WHERE

Ayuda a filtrar la lista de instancias retornadas.

hql> FROM Usuario user WHERE user.username = ‘rcuello’

hql> FROM Usuario user WHERE user.persona.cedula=123456

hql> FROM Usuario user WHERE username.username IS NOT NULL

Java Persistence Query Language (JPQL)

ORDER BY

La lista retornada por una consulta puede ser ordenada por cualquier propiedad de un objeto.

hql> FROM Empleado emp ORDER BY emp.nombre asc

hql> FROM Usuario user ORDER BY user.persona.apellido desc

GROUP BY

Def SQL : Combina los registros con valores idénticos, en la lista de campos especificados, en un único registro.

tienda	ganancia	fecha
Mcdonalds	1 200 000	05-Jan-1999
Kokoriko	950 000	07-Jan-1999
Mcdonalds	800 000	08-Jan-1999
American Brosted	720 000	08-Jan-1999

Tabla.
Tiendas_info

Deseamos saber las ventas totales para cada negocio. Para hacerlo, ingresaríamos,

SQL > SELECT tienda, SUM(ganancia) FROM Tiendas_info GROUP BY tienda

tienda	SUM (ganancia)
Mcdonalds	2 000 000
Kokoriko	950 000
Amrican Brosted	720 000

Dando como resultado

Java Persistence Query Language (JPQL)

GROUP BY

hql> Select t.tienda,SUM (t.ganancia) From TiendaInfo t GROUP BY t.tienda

El resultado de esta consulta es una Lista de Objects de tipo arreglo (Object[ ])
Ejemplo:

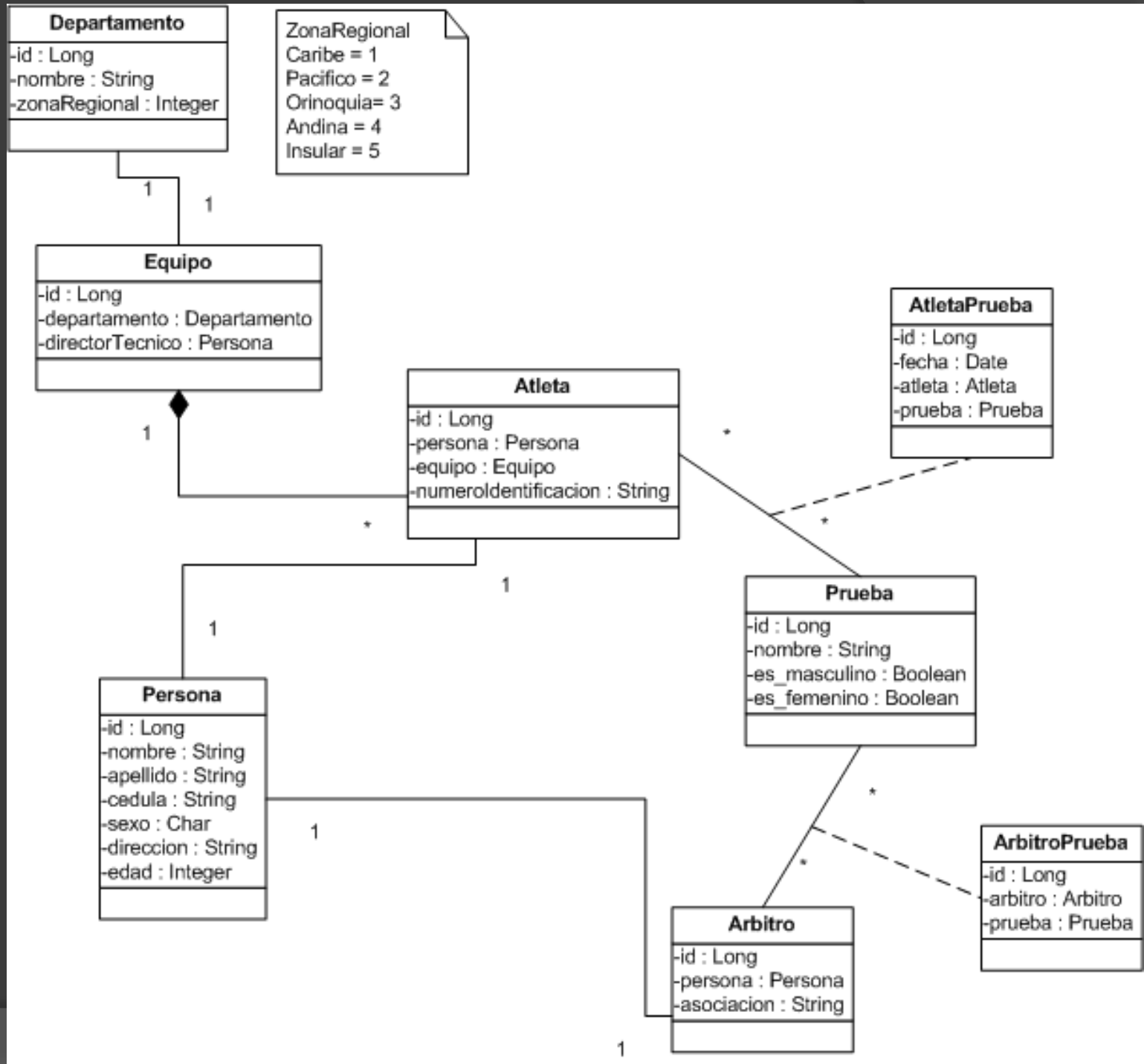
```
Query q=entityManager.createQuery("Select t.tienda,SUM (t.ganancia) From TiendaInfo t GROUP BY t.tienda ");
List lista =q.list();

For(Object obj : lista){
      Object[ ] listaObjetos=(Object[ ])obj;
      //donde la posicion 0 es igual a t.tienda
      //la posicion 1 es igual a SUM ( t.ganancia )
}
```

Java Persistence Query Language (JPQL)

Funciones predefinidas

Función	Definición
CONCAT(string1, string2)	Concatenates two string fields or literals
SUBSTRING(string, startIndex, length)	Returns the part of the string argument starting at startIndex (1-based) and ending at length characters past startIndex.
TRIM([LEADING | TRAILING | BOTH] [character FROM] string	Trims the specified character from either the beginning (LEADING) end (TRAILING) or both (BOTH) of the string argument. If no trim character is specified, the space character will be trimmed.
LOWER(string)	Returns the lower-case of the specified string argument.
UPPER(string)	Returns the upper-case of the specified string argument.
LENGTH(string)	Returns the number of characters in the specified string argument.
LOCATE(searchString, candidateString [, startIndex])	Returns the first index of searchString in candidateString. Positions are 1-based. If the string is not found, returns 0.
ABS(number)	Returns the absolute value of the argument.
SQRT(number)	Returns the square root of the argument.
MOD(number, divisor)	Returns the modulo of number and divisor.
CURRENT_DATE	Returns the current date.
CURRENT_TIME	Returns the current time.
CURRENT_TIMESTAMP	Returns the current timestamp.


Departamento
-id : Long
-nombre : String
-zonaRegional : Integer
ZonaRegional
Caribe = 1
Pacifico = 2
Orinoquia= 3
Andina = 4
Insular = 5
Equipo
-id : Long
-departamento : Departamento
-directorTecnico : Persona
AtletaPrueba
-id : Long
-fecha : Date
-atleta : Atleta
-prueba : Prueba
Atleta
-id : Long
-persona : Persona
-equipo : Equipo
-numeroIdentificacion : String
Prueba
-id : Long
-nombre : String
-es_masculino : Boolean
-es_femenino : Boolean
Persona
-id : Long
-nombre : String
-apellido : String
-cedula : String
-sexo : Char
-direccion : String
-edad : Integer
ArbitroPrueba
-id : Long
-arbitro : Arbitro
-prueba : Prueba
Arbitro
-id : Long
-persona : Persona
-asociacion : String
1
*

Gracias por su atención