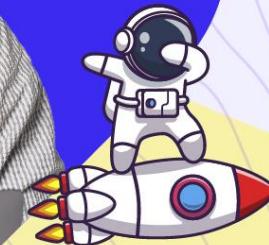
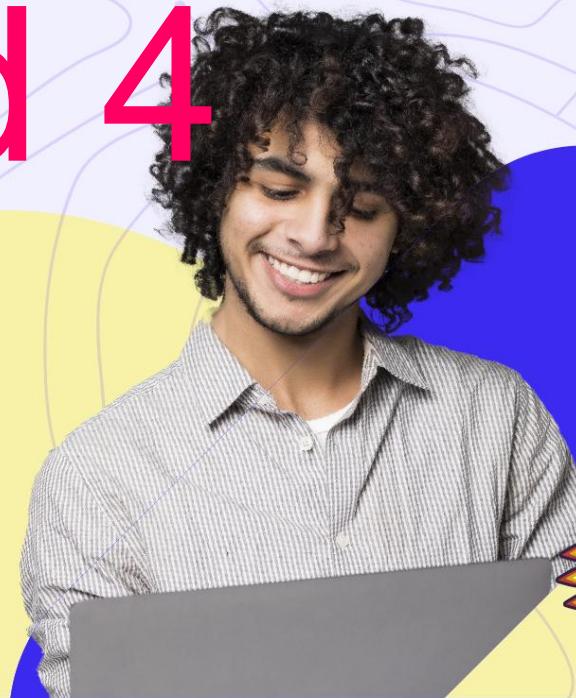




# Unidad 4

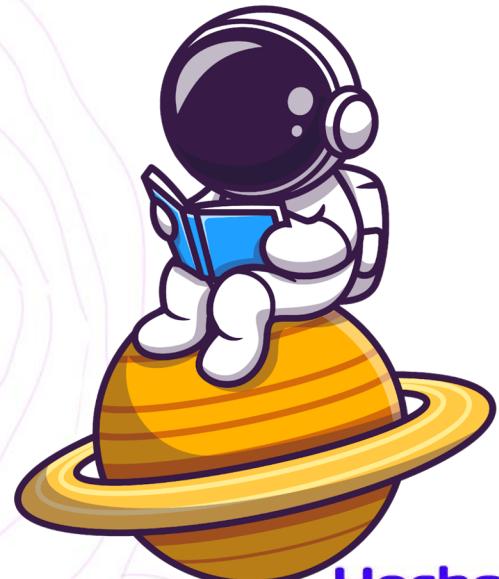
12 - ORM





# Persistencia en Java

- Se llama “persistencia” de los objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento.
  - Archivos
  - Bases de Datos Relacionales
    - JDBC - Java DataBase Connectivity
    - JPA - Java Persistence API





# JPA – Java Persistence API

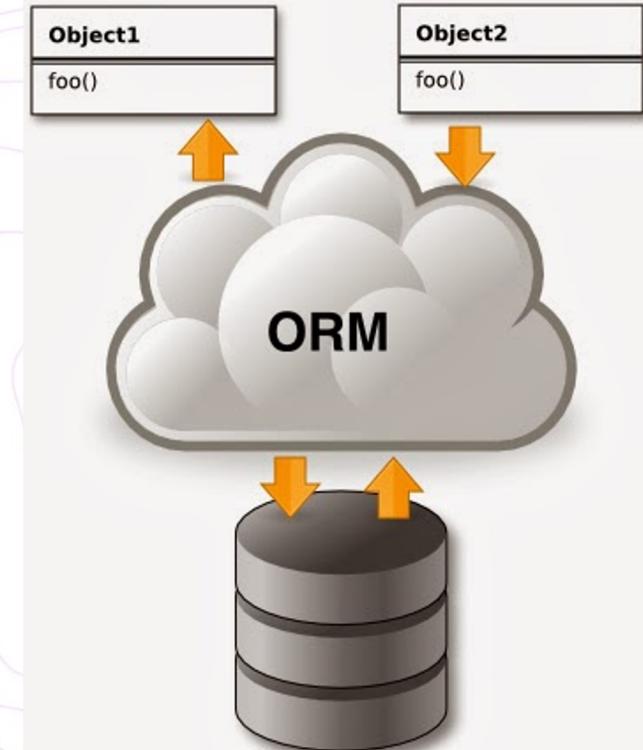
- Java Persistence API es la especificación interfaz de persistencia desarrollada para la gestión ORM con bases de datos relacionales, usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE).
- Persistencia en este contexto cubre tres áreas:
  - La API en sí misma, definida en el paquete javax.persistence
  - El lenguaje de consulta Java Persistence Query Language (JPQL).
  - Metadatos objeto/relacional.
- El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional), y permitir usar objetos regulares (conocidos como POJO).



# ORM - Object Relational Mapping

- El mapeo relacional de objetos (ORM) es una funcionalidad que se utiliza para desarrollar y mantener una relación entre un objeto y una base de datos relacional al mapear un estado de objeto a la columna de la base de datos.
- Es capaz de manejar varias operaciones de base de datos fácilmente, como insertar, actualizar, eliminar, etc.

Clase      ↔      Tabla  
Atributo    ↔      Campo





# Implementaciones JPA

- **Hibernate**
- **EJB3**
- **ObjectDB**
- **TopLink**
- **CocoBase**
- **EclipseLink**
- **OpenJPA**
- ...



**HIBERNATE**

**ORACLE**  
TOPLINK

eclipse)link

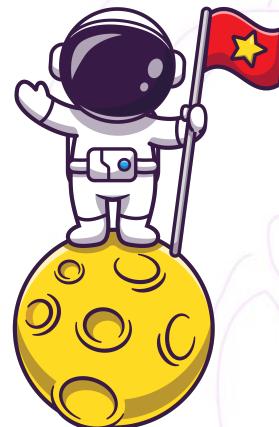
**EJB 3**

**OpenJPA**  
QUE CONECTAN ✓

**ObjectDB**



## 1. Agregar la dependencia **hibernate-core**



### pom.xml

```
...
<dependencies>
  ...
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.25.Final</version>
  </dependency>
  ...
</dependencies>
...
```



1. Agregar la dependencia  
**hibernate-core**
2. Agregar la dependencia del  
conector de base de datos



# Configurando JPA

## pom.xml

```
...  
<dependencies>  
...  
<dependency>  
  <groupId>org.xerial</groupId>  
  <artifactId>sqlite-jdbc</artifactId>  
  <version>3.36.0.1</version>  
</dependency>  
<dependency>  
  <groupId>com.github.gwenn</groupId>  
  <artifactId>sqlite-dialect</artifactId>  
  <version>0.1.2</version>  
</dependency>  
</dependencies>  
...
```



# SQLite Database URL

Tipo	Formato URL
Incrustada en la aplicación (Embebida)	<code>jdbc:sqlite:&lt;file_path&gt;</code> <code>jdbc:sqlite:sample.db</code> <code>jdbc:sqlite:C:/sqlite/db/sample.db</code>
En memoria	<code>jdbc:sqlite::memory:</code>



1. Agregar la dependencia hibernate-core
2. Agregar la dependencia del conector de base de datos
3. Creando la unidad de persistencia

# Configurando JPA

## META-INF/persistence.xml

```
<?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_2_1.xsd"
    version="2.1">

    <persistence-unit name="clase12-pu">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>

            <property name="javax.persistence.jdbc.driver"
                value="org.sqlite.JDBC" />
            <property name="javax.persistence.jdbc.url"
                value="jdbc:sqlite:sample.db" />
            <property name="javax.persistence.jdbc.username"
                value="" />
            <property name="javax.persistence.jdbc.password"
                value="" />

            <property name="hibernate.dialect"
                value="org.hibernate.dialect.SQLiteDialect" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.show_sql" value="true" />

        </properties>
    </persistence-unit>
</persistence>
```



# Anotaciones JPA

**@Entity:** Declarar que una clase será tratada como una entidad ORM

**@Table:** Declara las propiedades de una tabla de base de datos

**@Id:** Indica la clave primaria de una tabla

**@Column:** Permite declarar características de un campo de la tabla

**@JoinColumn:** Permite establecer relaciones entre entidades

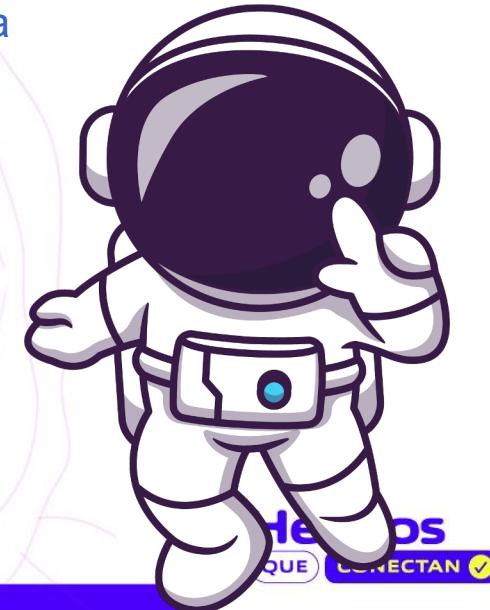
**@Temporal:** Especial trato de atributos de Fecha

**@ManyToOne:** Relación muchos a uno

**@OneToMany:** Relación uno a muchos

**@OneToOne:** Relación uno a uno

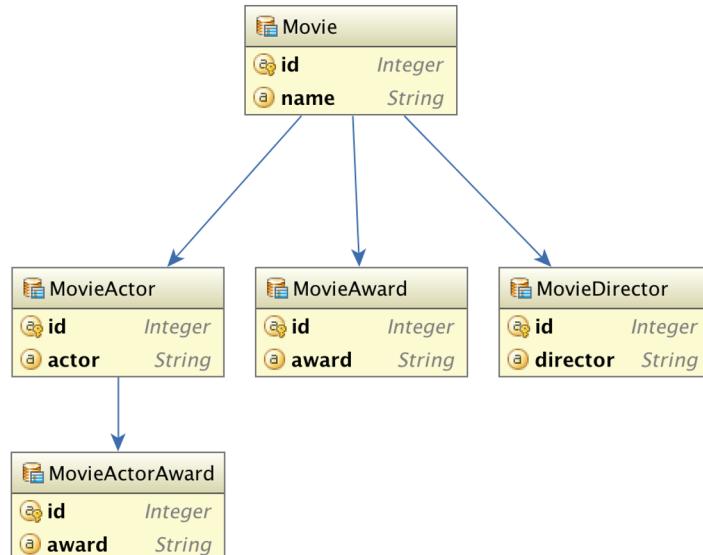
**@ManyToMany:** Relación de muchos a muchos





# Entidades (Entity)

- Una entidad de persistencia (**entity**) es una clase de Java ligera, cuyo estado es persistido de manera asociada a una tabla en una base de datos relacional.
- Las instancias de estas entidades corresponden a un registro (conjunto de datos representados en una fila) en la tabla.
- Normalmente las entidades están relacionadas a otras entidades, y estas relaciones son expresadas a través de metadatos objeto/relacional.



# Entidades (Entity)

```
@Entity
@Table(name = "Student")
public class Estudiante implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID")
    private Long id;

    @Column(name = "FNAME")
    private String nombres;

    @Column(name = "LNAME")
    private String apellidos;

    @Column(name = "CONTACT_NO")
    private String telefono;

    public Estudiante() {
        // Constructor obligatorio para JPA
    }

    // Setters y Getters
}
```

```
@Entity
@Table(name = "Student")
public class Persona implements Serializable {

    @Id
    private String nombre;
    private Integer edad;

    public Persona() {
        // Constructor obligatorio para JPA
    }

    public Persona(String nombre, int edad) {
        setEdad(edad);
        setNombre(nombre);
    }

    // Setters y Getters
}
```



# Mapeo de Tipos de Datos

Java Data Type	Database.com Data Type	Standard JPA Annotation
Boolean	Checkbox	N/A
Character	Text (255)	@Column(length=255)
Integer	Number (11, 0)	@Column(precision=11, scale=0)
Long	Number (18, 0)	@Column(precision=18, scale=0)
Double	Number (16, 2)	@Column(precision=16, scale=2)
Float	Number (16, 2)	@Column(precision=16, scale=2)
String	Text (255)	@Column(length=fieldLength)
java.util.Date	Date	@Temporal(TemporalType.DATE)
java.util.Date	Date/Time	@Temporal(TemporalType.TIMESTAMP)
BigInteger	Number (18, 0)	@Column(precision=18, scale=0)
BigDecimal	Currency	@Column(precision=16, scale=2)

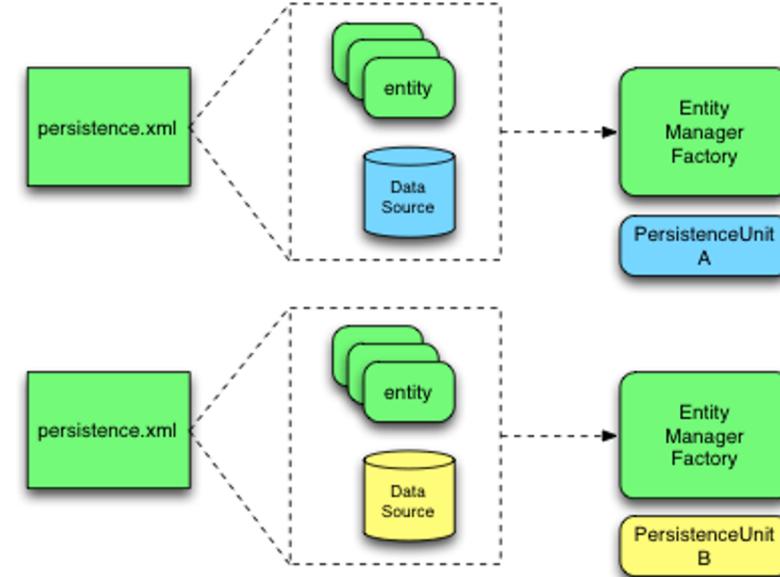


# EntityManagerFactory

En un primer lugar un **EntityManagerFactory** es único y es con el que nosotros gestionamos todas las entidades.

Si tenemos varias conexiones a base de datos deberemos definir un **PersistenceUnit** o unidad de persistencia, el cual se define en el archivo **persistence.xml**.

Cada PersistenceUnit tiene asociado un EntityManagerFactory diferente que gestiona un conjunto de entidades distinto.



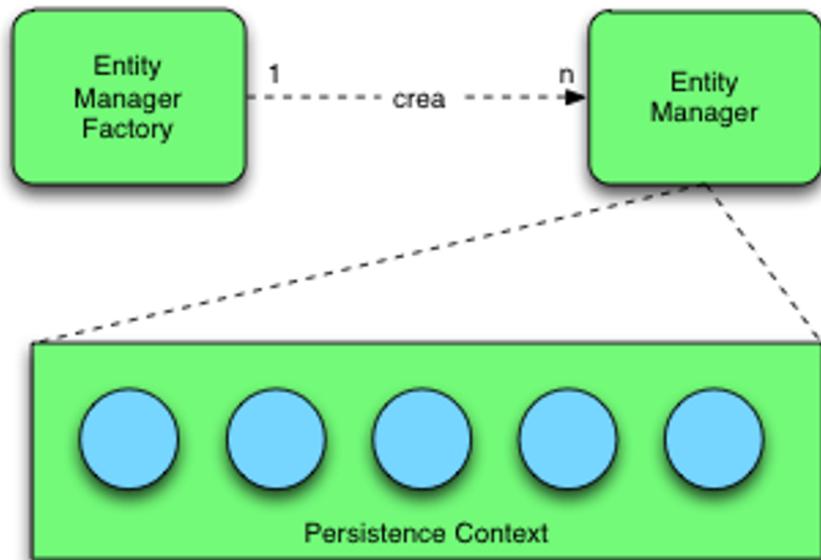


# EntityManager

Un EntityManagerFactory este será capaz de construir un objeto de tipo **EntityManager** que gestiona un conjunto de entidades u objetos.

Estas entidades son objetos POJO (Plain Old Java Object) normales con los cuales estamos trabajando en nuestro programa Java, que han sido etiquetadas para ser mapeadas con los objetos en la base de datos.

El **EntityManager** será el encargado de salvarlos a la base de datos, eliminarlos de la base de datos etc .



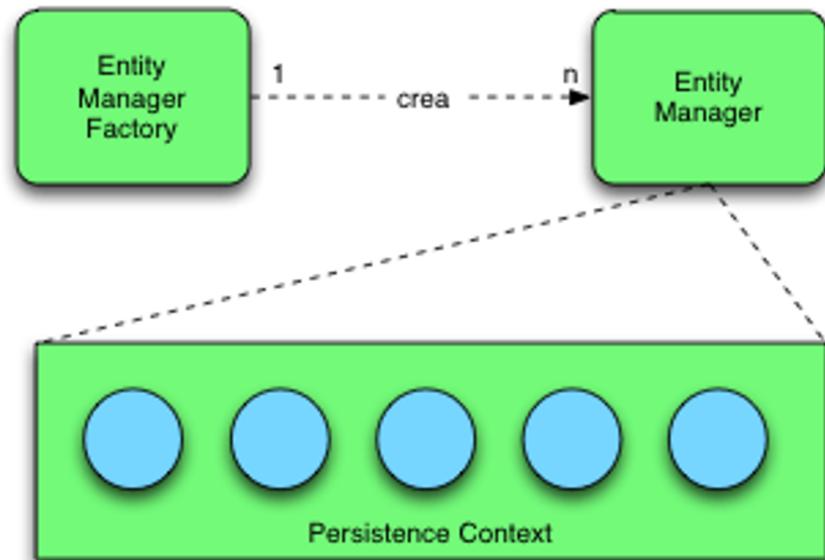


# EntityManager (métodos)

Un EntityManagerFactory este será capaz de construir un objeto de tipo **EntityManager** que gestiona un conjunto de entidades u objetos.

Estas entidades son objetos POJO (Plain Old Java Object) normales con los cuales estamos trabajando en nuestro programa Java, que han sido etiquetadas para ser mapeadas con los objetos en la base de datos.

El **EntityManager** será el encargado de salvarlos a la base de datos, eliminarlos de la base de datos etc .

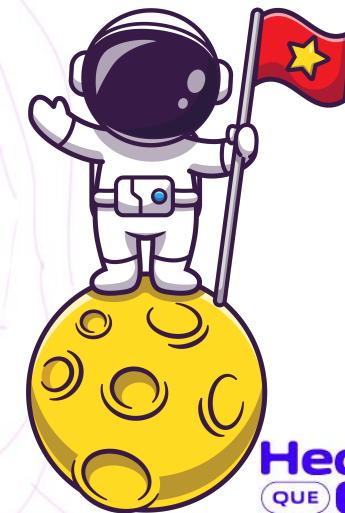




# Almacenar nuevas entidades: persist()

```
var yo = new Persona("Cesar Diaz", 41);

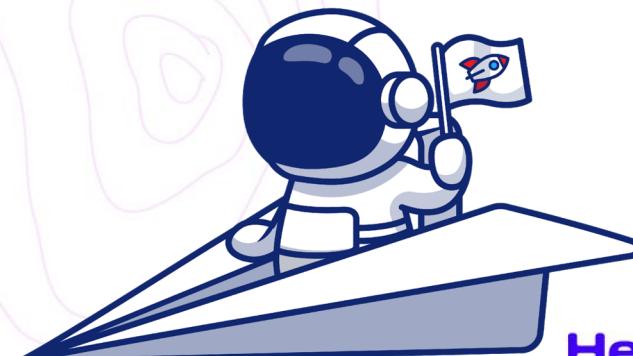
var emf = Persistence.createEntityManagerFactory("clase12-pu");
var em = emf.createEntityManager();
try {
    em.getTransaction().begin();
    em.persist(yo);
    em.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    em.close();
}
```





# Buscar una entidad por su clave: **find()**

```
var emf = Persistence.createEntityManagerFactory("clase12-pu");
var em = emf.createEntityManager();
try {
    var persona = em.find(Persona.class, "Cesar Diaz");
    System.out.println(persona.getNombre());
    System.out.println(persona.getEdad());
} catch (Exception e) {
    e.printStackTrace();
} finally {
    em.close();
}
```





# Eliminar una entidad: remove()

```
var emf = Persistence.createEntityManagerFactory("clase12-pu");
var em = emf.createEntityManager();
try {
    var yo = em.find(Persona.class, "Cesar Diaz");
    em.getTransaction().begin();
    em.remove(yo);
    em.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    em.close();
}
```





# Listar todas las entidades

La forma normal para hacer consultas personalizadas que devuelven más de 1 dato, es realizando una consulta con JPQL.

```
var emf = Persistence.createEntityManagerFactory("clase12-pu");
var em = emf.createEntityManager();
try {
    var query = em.createQuery("SELECT p FROM Persona p",
                               Persona.class);
    var personas = query.getResultList();
    personas.forEach(System.out::println);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    em.close();
}
```

Otra forma con la que podemos realizar la consulta es usando CriteriaAPI

```
var emf = Persistence.createEntityManagerFactory("clase12-pu");
var em = emf.createEntityManager();
try {
    var cb = em.getCriteriaBuilder();
    var cq = cb.createQuery(Persona.class);
    var rootEntry = cq.from(Persona.class);
    var all = cq.select(rootEntry);

    var query = em.createQuery(all);
    var personas = query.getResultList();
    personas.forEach(System.out::println);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    em.close();
}
```

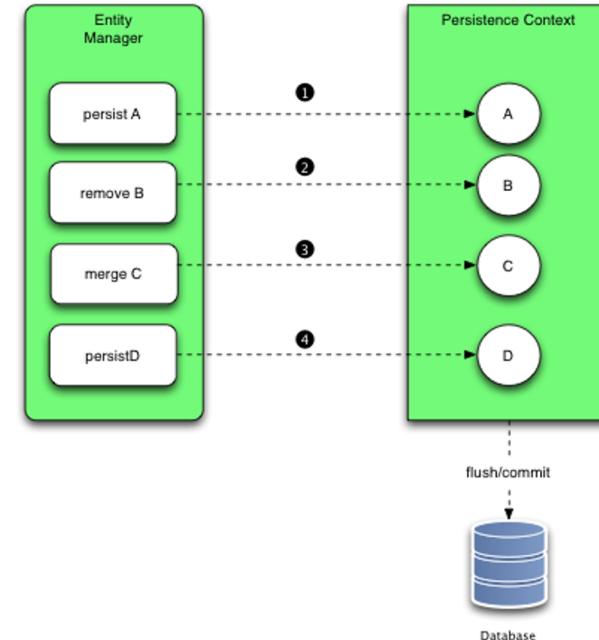


# Sincronizar cambios con la BD

El método **flush()** es un poco curioso y es el encargado de sincronizar el PersistenceContext contra la base de datos.

Normalmente todo el mundo piensa que cuando nosotros invocamos el método **persist()** o **remove()** se ejecutan automáticamente las consultas contra la base de datos.

Esto no es así ya que el EntityManager irá almacenando las modificaciones que nosotros realizamos sobre las entidades para más adelante persistirlas contra la base de datos todas de golpe ya sea invocando **flush** o realizando un **commit** a una transacción.





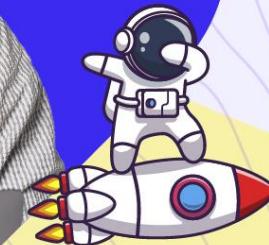
# Vamos al código

- Crear un proyecto Maven para los ejercicios de la clase 12
  - Ctrl + Shift + P
  - > Java: Create Java Project...
  - Maven
  - maven-archetype-quickstart, <versión más reciente>
  - group Id: co.edu.utp.misiontic2022.c2
  - artifact Id: clase12-jpa
- Realizar una copia de clase11-jdbc y realizar los ajustes que sean necesarios para convertirlo a JPA.





# Herencia en JPA





El futuro digital  
es de todos

Mision  
TIC

UTP  
Universidad Técnica  
de Pereira

Mision  
TIC 2022

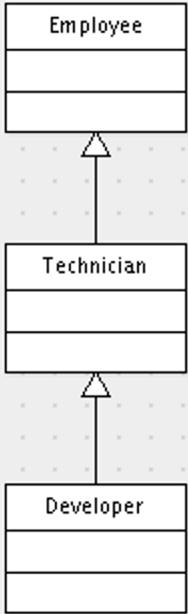
A man with a beard and short dark hair is speaking directly to the camera. He is wearing a black t-shirt. Behind him is a white surface with several silver cylindrical objects, possibly representing databases or server components. To his right is a large graphic element. This graphic features a green diagonal band on the right side. On the left part of the band, the letters "JPA" are written in large white capital letters. Below "JPA" is a stylized logo consisting of a red flame-like shape above three blue concentric arcs. Underneath this logo, the word "Java" is written in red lowercase letters. At the bottom of the green band, the words "Estrategias de Herencia" are written in large white capital letters.

Hechos  
QUE CONECTAN ✓



# Estrategias de mapeo en JPA

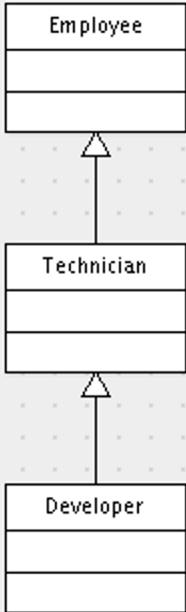
- Una sola tabla para guardar toda la jerarquía de clases. Tiene la ventaja de ser la opción que mejor rendimiento da, ya que sólo es necesario acceder a una tabla (está totalmente desnortualizada). Tiene como inconveniente que todos los campos de las clases hijas tienen que admitir nulos, ya que cuando guardemos un tipo, los campos correspondientes a los otros tipos de la jerarquía no tendrán valor.
- Una tabla para el padre de la jerarquía, con las cosas comunes, y otra tabla para cada clase hija con las cosas concretas. Es la opción más normalizada, y por lo tanto la más flexible (puede ser interesante si tenemos un modelo de clases muy cambiante), ya que para añadir nuevos tipos basta con añadir nuevas tablas y si queremos añadir nuevos atributos sólo hay que modificar la tabla correspondiente al tipo donde se está añadiendo el atributo. Tiene la desventaja de que para recuperar la información de una clase, hay que ir haciendo join con las tablas de las clases padre.
- Una tabla independiente para cada tipo. En este caso cada tabla es independiente, pero los atributos del padre (atributos comunes en los hijos), tienen que estar repetidos en cada tabla. En principio puede tener serios problemas de rendimiento, si estamos trabajando con polimorfismo, por los SQL JOINS que tiene que hacer para recuperar la información. Sería la opción menos aconsejable.



# SINGLE\_TABLE Strategy

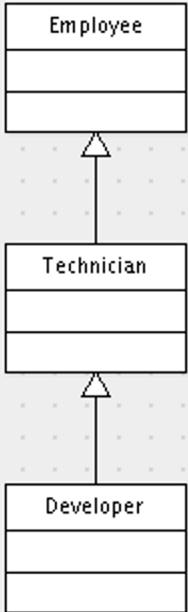
```
@Entity  
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name="type")  
public class Employee  
    implements Serializable {  
  
    @Id  
    @GeneratedValue  
    Long id;  
    private String nif;  
    private String name;  
    private String email;  
  
    public Employee() {  
    }  
  
    // Setters y getters  
}
```

```
@Entity  
@DiscriminatorValue("tech")  
public class Technician extends Employee {  
    private int experienceYears = 0;  
  
    // Setters y getters  
}  
  
@Entity  
@DiscriminatorValue("dev")  
@DiscriminatorColumn(name="type")  
public class Developer extends Technician {  
    private String expertLanguages = null;  
  
    // Setters y getters  
}
```



# SINGLE\_TABLE Strategy

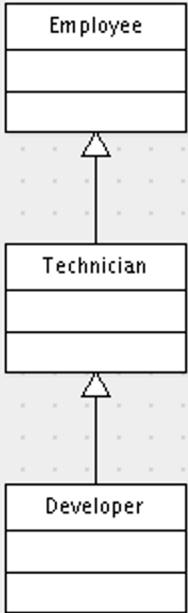
```
CREATE TABLE Employee (
    id integer NOT NULL,
    nif varchar(10) default NULL,
    name varchar(50) NOT NULL,
    phone varchar(20) default NULL,
    email varchar(50) default NULL,
    type varchar(31) default NULL,
    experienceYears int unsigned default 0,
    expertLanguages varchar(50) default NULL,
    PRIMARY KEY(id)
)
```



# JOINED Strategy

```
@Entity  
@Table(name = "Employee")  
@Inheritance(strategy=  
            InheritanceType.JOINED)  
public class Employee  
    implements Serializable {  
  
    @Id  
    @GeneratedValue  
    Long id;  
    private String nif;  
    private String name;  
    private String email;  
  
    public Employee() {  
    }  
  
    // Setters y getters  
}
```

```
@Entity  
@PrimaryKeyJoinColumn(name="employeeId")  
public class Technician extends Employee {  
    private int experienceYears = 0;  
  
    // Setters y getters  
}  
  
@Entity  
@PrimaryKeyJoinColumn(name="employeeId")  
public class Developer extends Technician {  
    private String expertLanguages = null;  
  
    // Setters y getters  
}
```

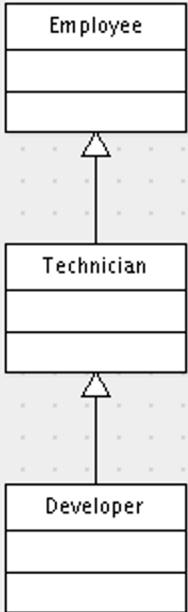


# JOINED Strategy

```
CREATE TABLE Employee (
    id integer NOT NULL,
    nif varchar(10) default NULL,
    name varchar(50) NOT NULL,
    phone varchar(20) default NULL,
    email varchar(50) default NULL,
    PRIMARY KEY (id),
);

CREATE TABLE Technician (
    id integer NOT NULL,
    employeeId integer NOT NULL,
    experienceYears integer default 0,
    PRIMARY KEY (id),
    foreign key (employeeId)
        references Employee (id)
);
```

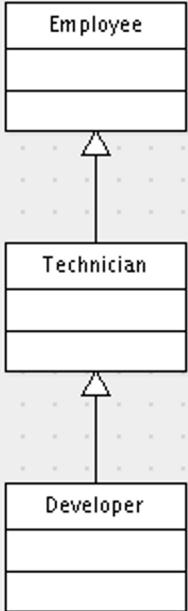
```
CREATE TABLE Developer (
    id integer NOT NULL auto_increment,
    employeeId integer NOT NULL,
    expertLanguages varchar(50),
    PRIMARY KEY (id),
    foreign key (employeeId)
        references Employee (id)
);
```



# TABLE\_PER\_CLASS Strategy

```
@Entity  
@Table(name = "Employee")  
@Inheritance(strategy=  
    InheritanceType.TABLE_PER_CLASS)  
public class Employee  
    implements Serializable {  
  
    @Id  
    @GeneratedValue  
    Long id;  
    private String nif;  
    private String name;  
    private String email;  
  
    public Employee() {  
    }  
  
    // Setters y getters  
}
```

```
@Entity  
@Table(name = "Technician")  
public class Technician extends Employee {  
    private int experienceYears = 0;  
  
    // Setters y getters  
}  
  
@Entity  
@Table(name = "Developer")  
public class Developer extends Technician {  
    private String expertLanguajes = null;  
  
    // Setters y getters  
}
```



# TABLE\_PER\_CLASS Strategy

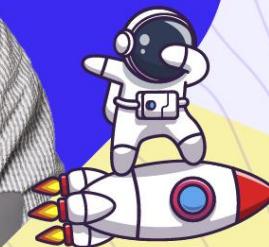
```
CREATE TABLE Employee (
    id integer NOT NULL,
    nif varchar(10) default NULL,
    name varchar(50) NOT NULL,
    phone varchar(20) default NULL,
    email varchar(50) default NULL,
    PRIMARY KEY (id)
);

CREATE TABLE Technician (
    id integer NOT NULL,
    nif varchar(10) default NULL,
    name varchar(50) NOT NULL,
    phone varchar(20) default NULL,
    email varchar(50) default NULL,
    experienceYears integer default 0,
    PRIMARY KEY (id),
    FOREIGN KEY (employeeId)
        references Employee (id)
);
```

```
CREATE TABLE Developer (
    id integer NOT NULL auto_increment,
    nif varchar(10) default NULL,
    name varchar(50) NOT NULL,
    phone varchar(20) default NULL,
    email varchar(50) default NULL,
    expertLanguages varchar(50),
    PRIMARY KEY (id),
    FOREIGN KEY (employeeId)
        references Employee (id)
);
```

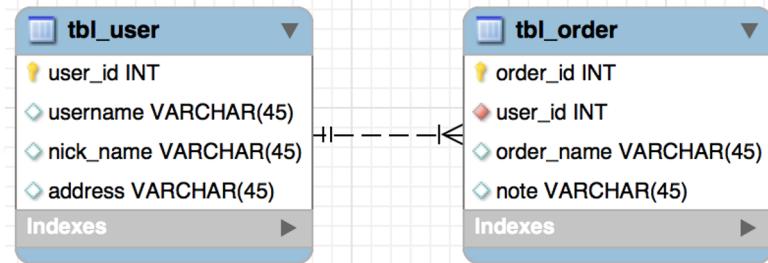


# Relaciones entre Entidades





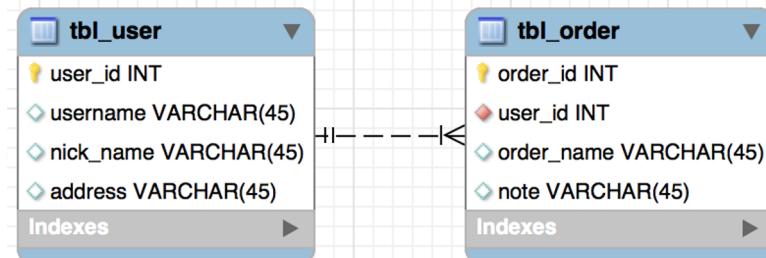
# @ManyToOne



- Existe relación **@ManyToOne** entre dos entidades donde se hace referencia a una entidad (columna o conjunto de columnas) con valores únicos que contienen de otra entidad (columna o conjunto de columnas).
- En bases de datos relacionales, estas relaciones se aplican mediante el uso de clave primaria clave externa entre las tablas.



# @OneToMany



- En esta relación, cada fila de una entidad se hace referencia a los muchos registros secundarios en otra entidad.
- Lo importante es que los registros secundarios no pueden tener varios padres.
- En una relación uno a varios entre la tabla A y B de la tabla, cada fila en la tabla A puede ser vinculado a una o varias filas en la tabla B.



# @ManyToOne y @OneToMany

```
@Entity
@Table("tbl_user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer userId;

    private String username;
    private String nickName;
    private String address;

    @OneToMany(mappedBy = "user",
               cascade = CascadeType.ALL)
    private List<Order> orders;

    // getter and setter

    public boolean addOrder(Order order) {
        if(orders == null)
            orders = new ArrayList<>();
        return this.orders.add(order);
    }

}
```

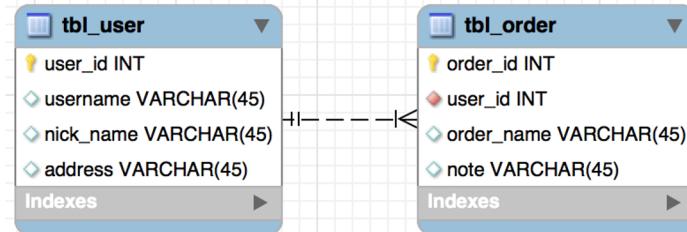
```
@Entity
@Table("tbl_order")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer orderId;

    private String orderName;
    private String note;

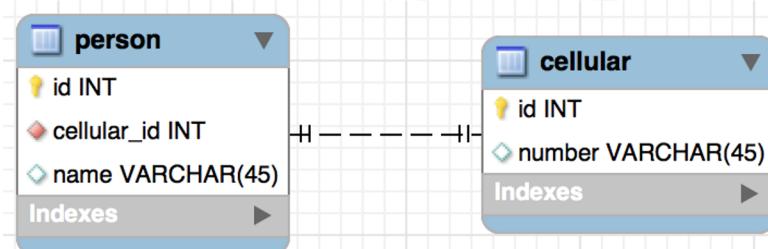
    @ManyToOne(fetch = FetchType.LAZY,
               cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id")
    private User user;

    // getter and setter
}
```





# @OneToOne



- En una relación **@OneToOne**, un elemento puede vincularse al único otro elemento. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.



# @OneToOne

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    public Person() {}

    public Person(String name,
                 Cellular cellular) {
        this.name = name;
        this.cellular = cellular;
    }

    @OneToOne
    @JoinColumn(name = "cellular_id")
    private Cellular cellular;

    // getter and setter
}
```

```
@Entity
public class Cellular {

    @Id
    @GeneratedValue
    private Integer id;

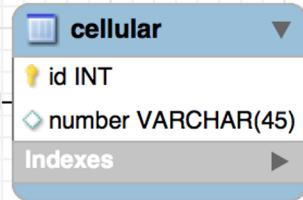
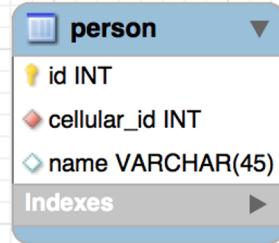
    private String number;

    public Cellular() {}

    public Cellular(int number) {

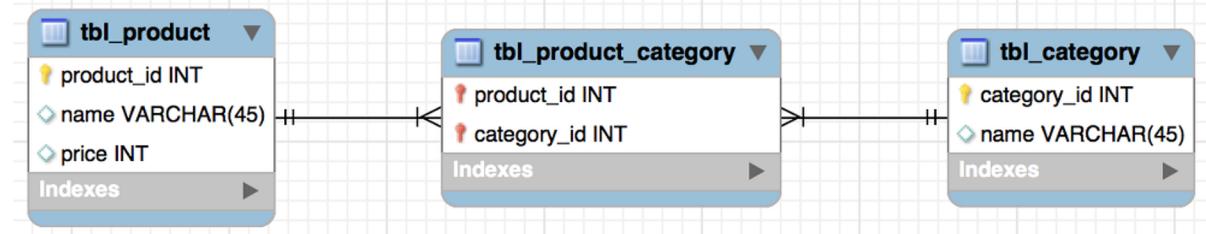
        this.number = number;
    }

    // getter and setter
}
```





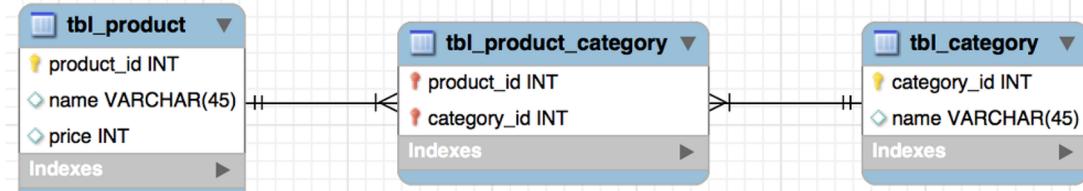
# @ManyToMany



- Una relación @ManyToMany es donde una o más filas de una entidad se asocian a más de una fila en otra entidad.



## @ManyToMany



```
@Entity
@Table("tbl_product")
public class Product {

    @Id @GeneratedValue
    private Integer productId;

    private String name;
    private int price;

    @ManyToMany(fetch = FetchType.LAZY,
               cascade = CascadeType.ALL)
    @JoinTable(name = "tbl_product_category",
               joinColumns=@JoinColumn(name="product_id"),
               inverseJoinColumns=@JoinColumn(name="category_id"))
    private List<Category> categories;

    // getter and setter

    public boolean addCategory(Category category) {
        if(categories == null)
            categories = new ArrayList<>();

        return this.categories.add(category);
    }
}
```

```
@Entity
@Table("tbl_category")
public class Category {

    @Id @GeneratedValue
    private Integer categoryId;

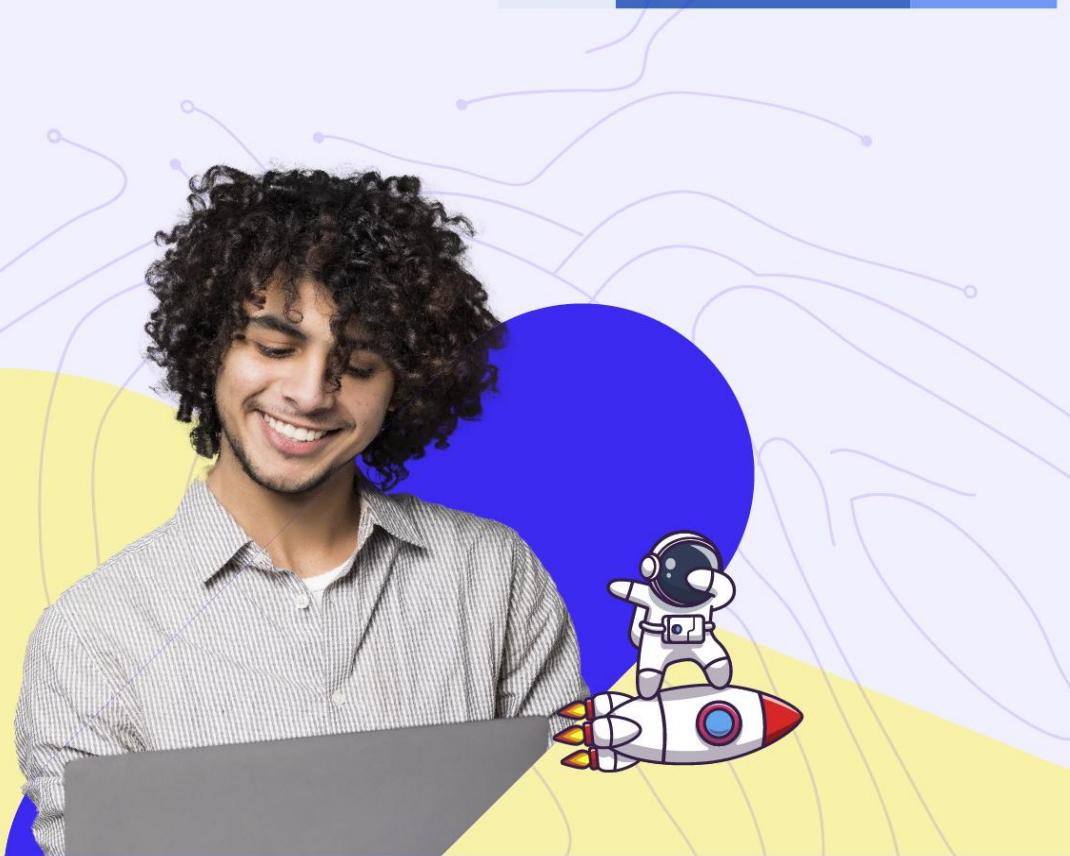
    private String name;

    @ManyToMany(fetch = FetchType.LAZY,
               mappedBy = "categories")
    private List<Product> products;

    // getter and setter
    public boolean addProduct(Product product) {
        if(products == null) {
            products = new ArrayList<>();
        }

        return this.products.add(product);
    }
}
```

# JPQL - JPA Query Language





# JPQL - Lenguaje de Consultas de JPA

JPQL es el lenguaje de consulta de persistencia de Java.

Se utiliza para crear consultas contra **entidades** para almacenar en una base de datos relacional. JPQL está desarrollado en base a la sintaxis SQL.

JPQL puede recuperar datos mediante la cláusula SELECT, puede hacer a granel actualizaciones con cláusula UPDATE y DELETE cláusula.



# Estructura de consulta

Sintaxis JPQL es muy similar a la sintaxis de SQL. Tener SQL como sintaxis es una ventaja porque SQL es simple y siendo ampliamente utilizado. SQL trabaja directamente contra la base de datos relacional tablas, registros y campos, mientras que JPQL trabaja con Java clases e instancias.

Por ejemplo, una consulta JPQL puede recuperar una entidad objeto en lugar de campo conjunto de resultados de una base de datos, al igual que con SQL.

```
SELECT ...  
FROM ...  
[WHERE ...]  
[GROUP BY ... [HAVING ...]]  
[ORDER BY ...]
```

```
DELETE FROM ...  
[WHERE ...]  
UPDATE ... SET ...  
[WHERE ...]
```



# Ejemplos de consultas JPA

Obtener todos los datos de la entidad Employee

```
SELECT e FROM Employee e
```

Obtener los nombres de todos los empleados en mayúscula

```
SELECT UPPER(e.ename) FROM Employee e
```

Obtener el salario máximo de la tabla Employee

```
SELECT MAX(e.salary) FROM Employee e
```

Obtener los empleados cuyo salario esté entre 10000 y 20000

```
SELECT e FROM Employee e  
WHERE e.salary BETWEEN 10000 AND 20000
```

Obtener los empleados cuyo nombre empieza con M mayúscula

```
SELECT e FROM Employee e  
WHERE e.firstName LIKE 'M%'
```

Obtener todos los empleados ordenados por nombre

```
SELECT e FROM Employee e ORDER BY e.firstName ASC
```

Obtener los datos de los empleados junto a la información del departamento al que pertenece

```
SELECT e, d FROM Employee e  
INNER JOIN e.Department d
```

Obtener los datos de todos los Países junto al nombre de su capital (si está asociada)

```
SELECT c, p.name FROM Country c  
LEFT OUTER JOIN c.capital p
```



# Para la próxima sesión...

- Terminar los ejercicios que no se terminaron... (si aplica)
- Revisar el funcionamiento de la aplicación que se encuentra en  
<https://github.com/cesardiaz-utp/MisionTIC2022-Ciclo2-Unidad4-JPA>
- Ver videos: (Material complementario)
  - JPA - Básico
  - JPA - Avanzado