



Tema 15: Persistencia de los objetos en bases de datos orientadas a objetos

¿Qué aprenderás?

- Conocer qué son las bases de datos orientadas a objetos.
- Realizar programas en Java que trabajan con bases de datos orientadas a objetos, insertando, modificando, borrando y consultando sus datos.

¿Sabías que...?

- El auge de las bases de datos orientadas a objetos sucede en la década de los 90.
- ODMG (Object Data Management Group) es un grupo de empresas que se encargan de desarrollar el modelo de objetos para la persistencia en las bases de datos orientadas a objetos.
- OQL (Object Query Language) es el lenguaje de consulta estándar para las bases de datos orientadas a objetos. Es el equivalente al lenguaje SQL en las bases de datos relacionales. Es un lenguaje tan complejo que aún ningún creador de software lo ha implementado correctamente.



15. Persistencia de los objetos en bases de datos orientadas a objetos

Las bases de datos orientadas a objetos se caracterizan porque no trabajan con los tipos de datos típicos que usan las bases de datos relacionales, sino que lo hacen con clases y objetos.

Este tipo de bases de datos surgieron para simplificar el proceso de guardado y recuperación de datos en los programas orientados a objetos. Éstos, cuando necesitan guardar información en una base de datos relacional, deben descomponer el objeto en datos simples, y viceversa cuando hay que leer de la base de datos.

Así pues, en una base de datos orientada a objetos se pueden guardar y recuperar directamente los objetos manteniendo su estructura.

15.1. Características de las bases de datos orientadas a objetos

Las bases de datos orientadas a objetos presentan las siguientes características:

- Se guardan objetos manteniendo su estructura.
- Cada objeto se identifica con el OID (Object Identifier) que asigna la base de datos.
- Se pueden guardar diferentes tipos de objetos. Incluso se puede mantener la información de objetos estructurados, es decir, objetos que tienen otro objeto como dato.
- Se pueden implementar conceptos como la herencia, el polimorfismo, la sobrecarga, etc.

Hay múltiples opciones para desarrollar bases de datos orientas a objetos como por ejemplo:

- ObjectDB , ObjectStore, db40, Jade, ObjectBox, ...

En este caso trabajaremos con ObjectDB, una ágil base de datos multiplataforma.



15.2. La base de datos ObjectDB

Es una base de datos orientada a objetos diseñada para trabajar con Java. Es un software multiplataforma diseñado para ser utilizado en SO con versiones de Java SE 5 o superior.

A diferencia de otras bases de datos, ObjectDB trabaja de forma nativa sobre los estándares de Java JPA (Java Persistence API) y JDO (Java Data Objects).

Para realizar consultas a la base de datos, ObjectDB soporta dos sintaxis:

- JPA Query Language (JPQL) : basado en la sintaxis SQL
- JPA Criteria : basado en la sintaxis Java

15.3. Vincular ObjectDB con nuestro proyecto

El link de descarga del archivo jar de ObjectDB lo encontrarás en el apartado de recursos y enlaces.

La descarga es un archivo .zip con ejemplos y herramientas para el desarrollo con ObjectDB. Explorando éste .zip encontramos dentro de la carpeta *bin* el archivo *objectdb.jar* y *explorer.jar*.

Para utilizar ObjectDB vamos a tener que incluir el *objectdb.jar* a nuestro proyecto tal y como se explica en el siguiente vídeo.



UF6_15_3_Vincular_ObjectDB_NetBeans



Para explorar el contenido de la base de datos ejecutaremos el archivo `explorer.jar` o bien lanzándolo con doble click o bien por línea de comandos accediendo desde terminal y ejecutando el comando:

```
java -jar explorer.jar
```

15.4. Creación/acceso a la base de datos ObjectDB

Podemos gestionar desde nuestros programas JAVA una base de datos a través de la clase `EntityManagerFactory`. Para poder mantener múltiples conexiones con una misma base de datos, `EntityManagerFactory` permite generar objetos del tipo `EntityManager` representando cada objeto una conexión con la base de datos.

De esta forma, siempre que queramos utilizar una base de datos en nuestro programa, primero vamos a tener que crear el objeto `EntityManagerFactory` y vincularlo con la base de datos indicando su ubicación. Y a partir de ése objeto utilizaremos su función `createEntityManager()` para crear tantos objetos como conexiones queramos con la base de datos:

```
// Vinculamos con la base de datos con el objeto emf
EntityManagerFactory emf = Persistence.createEntityManagerFactory("objectdb:db/ejemploDB.odb");
// creamos una conexión con la base de datos:
EntityManager em = emf.createEntityManager();
try {
    //Aquí las operaciones con la base de datos
} finally {
    // cerramos la conexión con la base de datos
    em.close();
    // cerramos el vínculo con la base de datos
    emf.close();
}
```

Creación/acceso a una base de datos en ObjectDB

Si la base de datos no existe, al crear el vínculo se crea.

Es muy importante cerrar cada conexión y el vínculo con la base de datos después de utilizarla. No vamos a poder explorar la base de datos con `explorer.jar` si no se ha cerrado el vínculo. Y de igual manera no vamos a poder vincularnos con la base de datos si la estamos explorando desde `explorer.jar`.

Para cerrar la el vínculo y la conexión con la base de datos utilizaremos la función. `close()` de `EntityManagerFactory` y `EntityManager`.



Para crear un vínculo a la base de datos especificando un usuario y contraseña, debemos añadir a la URL los parámetros *user* y *password*. Por ejemplo, si queremos que las conexiones con ejemploDB.odt se realicen con el usuario “admin” y el password “admin”, escribiremos:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory (
    "objectdb:db/ejemploDB.odt;user=admin;password=admin");
```

Podemos borrar todo el contenido existente en la base de datos añadiendo el parámetro *drop*. Como medida de seguridad, el parámetro *drop* solo funciona con bases de datos con la extensión .tmp. En el siguiente ejemplo se borran el contenido de *ejemploDB.tmp* al vincularse la base de datos:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory (
    "objectdb:db/ejemploDB.tmp;drop");
```

15.5. Guardar datos en ObjectDB

En las bases de datos orientadas a objetos, las “tablas” que tendríamos en una base de datos relacional pasarán a ser clases formadas por sus atributos. Siguiendo la misma comparación, los registros de cada tabla pasaran a ser distintas instancias de la clase.

Cuando queramos guardar un dato definiremos las clases para estructurar la información. Las instancias de estas clases serán los valores que finalmente guardaremos en la base de datos.

No todas las clases pueden ser guardadas en una base de datos orientada a objetos. ObjectDB permite almacenar:

- Tipos propios: Entity Clases, Mapped superclases, Embeddable clases.
- Tipos simples: Primitivas, Wrappers, String, Date, tipos de Math.
- Múltiples valores: Colecciones, Mapas y Arrays
- Mixtos: Tipos Enum y serializables

15.5.1. Crear Entidades

JPA Entity Classes son las clases propias creadas para estructurar la información que queremos guardar. Las instancias de estas clases contendrán los valores que finalmente guardaremos en la base de datos.

Las clases que queramos utilizar como estructura de la base de datos se crean como clases simples precedidas con la notación `@Entity`. El significado de ésta notación se tiene que importar desde `javax.persistence.*`; Esta notación indica que la clase ha de poder estar mapeada a una base de datos.

Los atributos que queramos guardar serán los que indiquemos en la clase. En la base de datos solo se almacenarán atributos, nunca los métodos.



Los atributos con el modificador de acceso *static*, *final*, *transient* o con la notación *@Transient* no se almacenarán en la base de datos.

Para indicar que un atributo es clave, precedimos la declaración del atributo con la notación *@Id*. Para indicar que un atributo queremos que sea autogenerado por la base de datos, precedimos la declaración del atributo con la notación *@GeneratedValue*.

Podemos combinar ambas notaciones para indicar un atributo clave y autogenerado:

```
@Id @GeneratedValue long id;
```

En el siguiente ejemplo creamos la clase *Persona* para almacenar el nombre y la edad de una persona. Como atributo clave y autogenerado creamos el atributo *id* para que cada instancia de *Persona* tenga un valor *id* distinto. Todos los atributos los creamos con el modificador de acceso *private* y creamos los getters y setters que nos interesen (por ejemplo, no nos interesa un setter de *id* ya que lo establece la base de datos). El ejemplo que se ha sobrescrito la función *toString()* para simplificar la salida de datos.



```
package javaproyectosobjectdb;
import javax.persistence.*;
@Entity
public class Persona {
    @Id @GeneratedValue long id;
    private String nombre;
    private int edad;
    //constructor
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    //Getters/Setters
    public long getId() { return id; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public int getEdad() { return edad; }
    public void setEdad(int edad) { this.edad = edad; }

    @Override
    public String toString(){
        return "ID:"+this.id
            +"Nombre:"+getNombre()
            +"Edad:"+getEdad();
    }
}
```

clase Persona para estructurar la información



15.5.2. Relacionar Entidades

Cuando una entidad contiene como un atributo a otra clase persistente, se produce una relación. Marcamos los atributos que relacionan dos clases con cuatro posibles notaciones que indicarán el tipo de relación:

- `@ManyToMany` : una relación de N a N.
- `@ManyToOne` : una relación de N a 1.
- `@OneToMany`: una relación de 1 a N.
- `@OneToOne`: una relación de 1 a 1 (No es necesario indicarla, relación por defecto).

Cuando la relación es de uno a varios, la clase con relación N ha de contener un atributo de la clase 1 con la notación `@ManyToOne` para vincular ambas clases. La clase con la relación 1 ha de contener una lista para almacenar la clase N con la notación `@OneToMany` y la propiedad `mappedBy` especificando el nombre del atributo de la clase N al que le hemos dado la notación `@ManyToOne`.

Aún y estando los campos relacionados, por defecto no se propagará ninguna operación. Para que se propaguen todos los cambios que se produzcan entre clases relacionadas añadimos la propiedad `cascade=CascadeType.ALL`, `orphanRemoval = true`.

El siguiente ejemplo crea una clase `Libreria` con una lista de libros relacionados con la notación:

```
@OneToMany(cascade=CascadeType.ALL,mappedBy="libreria",orphanRemoval = true)
```

La clase `Libro` se relaciona con `Libreria` con el atributo de clase `Libreria` , nombre `libreria` y notación `@ManyToOne`



```
package tema15M03II_ejemploRelaciones;

import javax.persistence.*;
import java.util.*;

@Entity
public class Libreria {
    @Id @GeneratedValue long id;
    String direccion;
    //con mappedBy relacionamos el atributo librería de la clase Libro con ésta clase
    @OneToMany(cascade=CascadeType.ALL,mappedBy="libreria",orphanRemoval = true)
    private List<Libro> libros= new ArrayList<>();

    public Libreria(String direccion ) {
        this.direccion = direccion;
    }
    public void addLibro(Libro l){
        l.setLibreria(this); //importante vincular la librería con el libro al añadirlo
        this.libros.add(l);
    }
    @Override
    public String toString() {
        return "Libreria{" + "id=" + id + ", direccion=" + direccion + ", libros=" + libros + '}';
    }
}
```

Ejemplo de relación **OneToMany** con la clase Libreria conteniendo varios Libro



```
package tema15M03II_ejemploRelaciones;

import javax.persistence.*;

@Entity
public class Libro {
    @Id @GeneratedValue
    long id;
    String titulo;

    @ManyToOne //relacionamos ésta clase con Libreria
    private Libreria libreria;

    public Libro(String titulo) {
        this.titulo = titulo;
    }
    //para terminar la relación, al crear el libro le hemos de asignar la librería
    public void setLibreria(Libreria libreria) {
        this.libreria = libreria;
    }
    @Override
    public String toString() {
        return "Libro{" + "id=" + id + ", titulo=" + titulo + '}';
    }
}
```

Ejemplo de relación **ManyToOne** en el que muchos Libro pueden estar relacionados con una única Libreria



15.5.3. Guardar Entidades

Cualquier operación que modifique el estado de la base de datos (crear, modificar o eliminar) deberemos ejecutarla dentro de una transacción.

Cada conexión solo puede mantener abierta una única transacción a la vez. Por ello, es cada objeto del tipo EntityManager (cada conexión) el encargado de iniciar y finalizar sus transacciones con las respectivas funciones:

`.getTransaction().begin();`

`.getTransaction().commit();`

Una vez iniciada una transacción, cuando queramos guardar un dato utilizaremos el objeto de tipo EntityManager para guardar la instancia de un objeto con la función:

`.persist(objeto_a_guardar);`

En el siguiente código, vamos a guardar en la base de datos dos instancias de la clase Persona creada en el punto **¡Error! No se encuentra el origen de la referencia.** .

```
// Vinculamos con la base de datos con el objeto emf
EntityManagerFactory emf = Persistence.createEntityManagerFactory("objectdb:db/ejemploDB.oddb");
// creamos una conexión con la base de datos:
EntityManager em = emf.createEntityManager();
try {
    //creamos las instancias
    Persona per1 = new Persona("Juana",78);
    Persona per2 = new Persona ("Jun",45);
    //iniciamos la transacción
    em.getTransaction().begin();
    //insertamos los datos en la base de datos
    em.persist(per1);
    em.persist(per2);
    //finalizamos y enviamos la transaccion
    em.getTransaction().commit();
} finally {
    // cerramos la conexión con la base de datos
    em.close();
    // cerramos el vínculo con la base de datos
    emf.close();
}
```

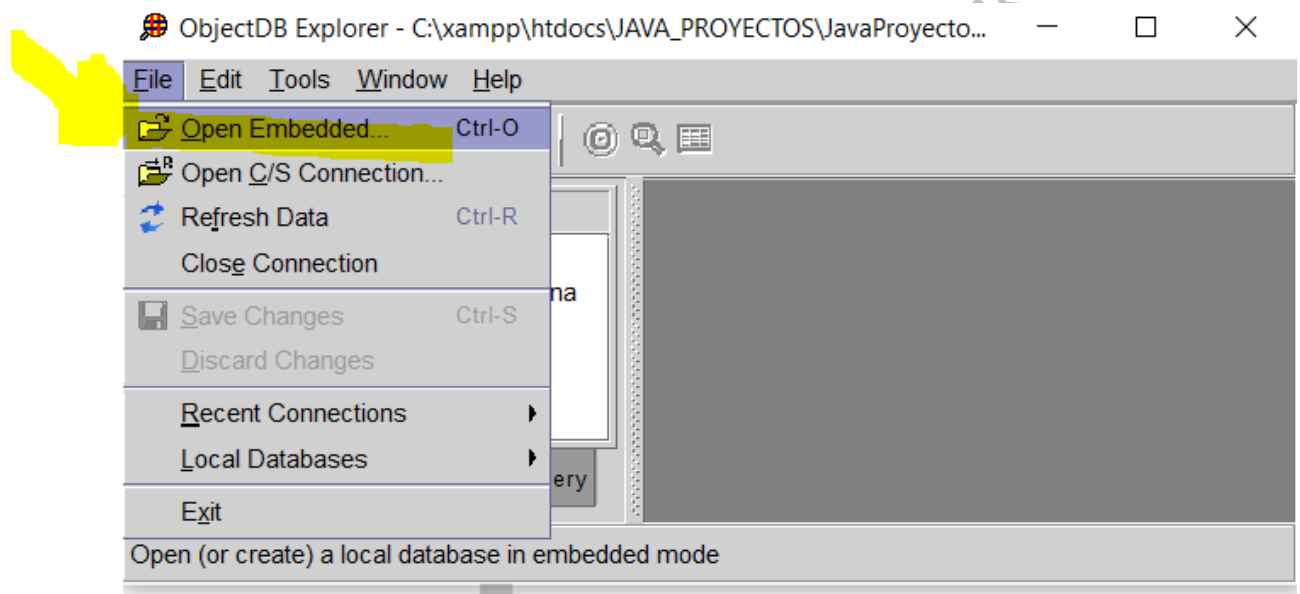
Creamos y guardamos dos personas en la base de datos



Con el ejemplo anterior, cada vez que ejecutamos nuestro programa añadimos dos personas más a la base de datos. Pero cuando desarrollamos es muy común querer ejecutar y limpiar la base de datos cada vez que se ejecuta el programa. Para conseguirlo tendremos que crear una base de datos con la extensión *.tmp* y abrirla con el parámetro *drop* tal como vimos en el punto 15.4 .

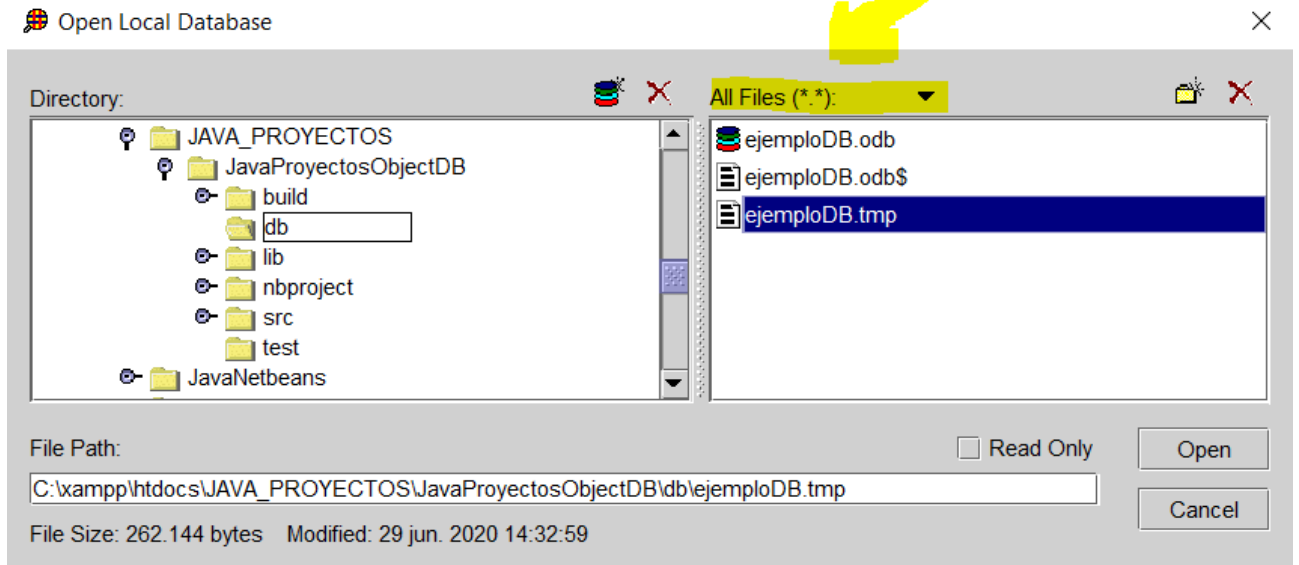
Una vez creada la base de datos e insertados los primeros registros, podemos explorar la base de datos ejecutando *explorer.jar*.

Para poder abrir una base de datos clicamos en File -> Open Embedded...




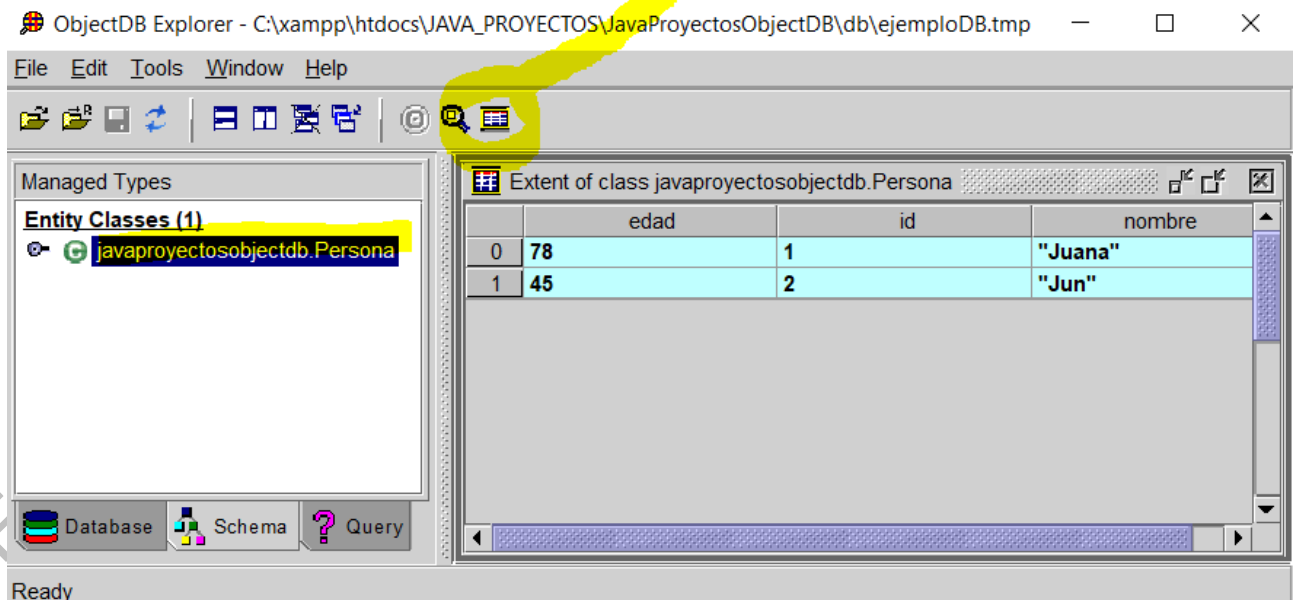
Abrir una base de datos .odb con explorer.jar

Seguidamente tenemos que seleccionar la base de datos. Para ver las bases de datos temporales, clicamos en ODB Files (*.odb) y seleccionamos ALL Files(*.*). Sino solo se muestran los archivos. odb.



Cambiamos el filtro para mostrar todos tipos de archivos y poder seleccionar una DB temporal.

Por último, seleccionamos la clase de la que queremos ver las instancias guardadas, y clicamos en  para ver los valores almacenados en la base de datos.



Mostramos los valores almacenados de Persona.



El siguiente ejemplo utiliza las clases Libro y Librería creadas en 15.5.2 para crear una librería con dos libros y guardarlo en la base de datos de nombre *reIDB.tmp*.

```
package tema15M03II_ejemploRelaciones;
import javax.persistence.*;
public class GestionLibros {
    public static void main(String[] args) throws Exception {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("objectdb:db/reIDB.tmp;drop");
        EntityManager em = emf.createEntityManager();

        Libreria libreria = new Libreria("mi calle");
        libreria.addLibro(new Libro("E.T."));
        libreria.addLibro(new Libro("Viven"));

        em.getTransaction().begin();
        em.persist(libreria);
        em.getTransaction().commit();
    }
}
```

Guardamos en la base de datos dos clases vinculadas.

15.6. Recuperar datos de la base de datos

ObjectDB permite realizar consultas a la base de datos mediante dos sintaxis:

- JPA Query Language (JPQL) : basado en la sintaxis SQL
- JPA Criteria API Queries: basado en la sintaxis Java

En ambos casos las consultas sobre la base de datos retornaran el objeto (o lista de objetos) que cumplan los criterios de la búsqueda.

En ningún caso es necesario realizar la consulta dentro de una transacción, ya que una consulta no modifica la base de datos.



15.7. JPQL

Java Persistence Query Language (JPQL) es un lenguaje muy similar a SQL con la diferencia principal que SQL trabaja con tablas y registros y en cambio JPQL trabaja con clases y objetos.

Su estructura básica para una consulta de selección sigue el siguiente formato:

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

Estructura básica de una consulta de selección JPQL.

Las dos primeras cláusulas, SELECT y FROM , son obligatorias. Mientras que WHERE, GROUP BY, HAVING y ORDER BY son opcionales.

La estructura básica para eliminar y actualizar contenido sigue el siguiente formato:

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

Estructura básica de una consulta JPQL par eliminar o actualizar .

15.7.1. Cláusula SELECT y consultas básicas

Para ejecutar una consulta con una sintaxis SQL primero creamos la consulta y posteriormente la ejecutamos obteniendo la respuesta. En ambos casos utilizamos el objeto *EntityManager* correspondiente.

Para crear la consulta utilizamos el método. `createQuery()` pasando como primer parámetro la consulta SQL y como segundo parámetro el tipo de datos a retornar por la consulta. De ésta forma, el método `createQuery()` retorna un objeto del tipo *TypedQuery* que utilizaremos para ejecutar la consulta.

Por ejemplo, para crear una Query que retorne todas las personas:

```
TypedQuery<Persona> q2 = em.createQuery("SELECT p FROM Persona p", Persona.class);
```



El lenguaje JPQL no permite utilizar la expresión "SELECT *", siempre hay que indicar específicamente los campos a obtener.

Una vez creada la consulta, el método *getResultList()*; retorna una lista de todos los objetos que cumplan la consulta.

Por ejemplo, para listar todas las personas:

```
TypedQuery<Persona> query = em.createQuery("SELECT p FROM Persona p", Persona.class);
List<Persona> personas = query.getResultList();
for (Persona p : personas) {
    System.out.println(p);
}
```

Consulta JPQL para obtener todas las personas.

Cuando nuestra consulta solo pueda retornar un único resultado, podemos utilizar el método *getSingleResult()*. Este método puede lanzar principalmente dos excepciones:

- *NonUniqueResultException*: esta excepción se lanza cuando la consulta genera más de un resultado.
- *NoResultException*: esta excepción se lanza cuando la consulta no genera ningún resultado.

Las excepciones derivadas de *getSingleResult()* han de ser gestionadas por el programador.

```
TypedQuery<Persona> q1 = em.createQuery("SELECT p FROM Persona p WHERE nombre='Juana' ",
Persona.class);
try {
    Persona p1 = q1.getSingleResult();
} catch (NonUniqueResultException e1) {
    System.out.println("Error, el id no es unico");
} catch (NoResultException e2) {
    System.out.println("No existe ningún grupo con el id "+idGrupo);
}
System.out.println(p1);
```

Consulta JPQL para obtener una persona.



Si seleccionamos múltiples campos obtendremos por cada resultado un array de Object que contendrá en cada posición cada uno de los campos.

En el siguiente ejemplo se seleccionan los campos de nombre y edad guardándolos en la posición 0 y 1 respectivamente de una Object []

```
TypedQuery<Object[]> query = em.createQuery("SELECT p.nombre, p.edad FROM Persona p", Object[].class);  
List<Object[]> personas = query.getResultList();  
for (Object[] p : personas) {  
    System.out.println("nombre:"+p[0]+" edad:"+p[1]);  
}
```

Consulta JPQL para obtener una persona.

Alternativamente también podemos guardar el resultado de una selección de múltiples campos en una clase que contenga los distintos campos como atributos. Para especificar una clase añadimos después del SELECT la palabra clave NEW seguido de la ruta completa a la clase y entre paréntesis los distintos atributos separados por comas. Para que la clase sea válida, ha de tener definido un constructor público que reciba los atributos.

En el siguiente ejemplo creamos dentro del package *ejpack* una clase *NombreEdad* solamente con el nombre y la edad para obtener la respuesta.

```
package ejpack;  
  
public class NombreEdad {  
    public String nombre;  
    public int edad;  
    public NombreEdad(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

Clase para estructurar la respuesta.



A continuación creamos la consulta añadiendo *NEW ejpack.NombreEdad(p.nombre, p.edad)* para indicar que el resultado lo obtendremos como una clase *NombreEdad* creada con un constructor que reciba el nombre y la edad.

```
TypedQuery<NombreEdad> query = em.createQuery("SELECT NEW ejpack.NombreEdad(p.nombre, p.edad)
FROM Persona AS p", NombreEdad.class);
List<NombreEdad> personas = query.getResultList();
for (NombreEdad p : personas) {
    System.out.println("nombre:"+p.nombre+" edad:"+p.edad);
}
```

Consulta JPQL para obtener como respuesta una nueva clase *NombreEdad*.

15.7.2. Cláusula FROM

Con cada consulta SELECT se itera sobre un conjunto de objetos comprobando si cada uno de los objetos satisface la cláusula WHERE. Con la cláusula FROM definimos los objetos sobre los que vamos a iterar en la consulta. Para ello escribimos FROM seguido de la clase sobre la que vamos a iterar seguido del nombre del objeto que con el contenido de cada iteración.

Por ejemplo, la consulta "SELECT per FROM Persona per " nos indica que iteraremos sobre la clase Persona guardando el valor de cada iteración en el objeto "per".

Es posible iterar sobre distintas clases con FROM separándolas con una coma. En el siguiente ejemplo iteramos sobre dos clases Persona para obtener aquellas personas con la misma edad. Para evitar valores duplicados, hemos descartado aquellas personas con el mismo nombre:

```
TypedQuery<Persona> query = em.createQuery("SELECT DISTINCT NEW Persona(p1.nombre,p2.edad)
FROM Persona p1, Persona p2 WHERE p1.edad = p2.edad and p1.nombre <> p2.nombre", Persona.class);
List<Persona> personas = query.getResultList();
for (Persona p : personas) {
    System.out.println(p);
}
```

Consulta con un FROM con múltiples clases.



15.7.3. Cláusula JOIN en JPQL

La cláusula JOIN la debemos utilizar seleccionar valores de una lista contenida en una tabla. En el siguiente código , siguiendo el ejemplo de Libro y Librerías en el apartado 15.5.3 , se utiliza la cláusula JOIN para extraer todos los libros almacenados en la librería con id "1".

```
//Mostrar los libros
TypedQuery<Libro> query = em.createQuery("SELECT ls FROM Libreria la JOIN la.libros ls WHERE la.id=1",
Libro.class);
List<Libro> libros = query.getResultList();
System.out.println("Recuperados " + libros.size() + " integrantes ");
for (Libro i : libros) {
    System.out.println(i);
}
```

Consulta con un JOIN entre una clases y su atributo de tipo lista.

15.7.4. Cláusula WHERE en JPQL

Para programar una consulta con valores del WHERE que sean variables, es recomendable utilizar parámetros en vez de crear la consulta con variables. La razón, es que utilizar variables para crear la consulta implica tener que crear una consulta distinta por cada valor distinto. En cambio, utilizando parámetros solo creamos una consulta optimizando de esta forma el rendimiento. A demás, crear la consulta con variables es menos seguro que utilizar parámetros, ya que expone la consulta a ataques de JPQL injection.

En una consulta JPQL indicamos que un valor será un parámetro escribiendo dos puntos seguido del nombre del parámetro. Antes de ejecutar la consulta asignamos el valor del parámetros con el método *setParameter()* que proporciona *TypedQuery*.

En el siguiente ejemplo primero concatenamos la variable *name* para crear una consulta variable y después creamos la misma consulta con el parámetro *nom* .



```
String name = "Juana";  
//concatenando la variable con la string (creará una consulta nueva cada vez)  
TypedQuery<Persona> q1 = em.createQuery("SELECT p FROM Persona p WHERE nombre='"+name+"'",  
Persona.class);  
Persona p1 = q1.getSingleResult();  
System.out.println(p1);  
  
//Utilizando el parametro :nom (podremos reaprovechar la consulta asignando nuevos parámetros)  
TypedQuery<Persona> q1p =  
    em.createQuery("SELECT p FROM Persona p WHERE nombre= :nom ", Persona.class);  
q1p.setParameter("nom", name); //establecemos el valor del parámetro nom  
Persona p1p = q1p.getSingleResult();  
System.out.println(p1p);
```

Distintas formas de utilizar consultas con parámetros variables.

15.7.5. Cláusula GROUP BY en JPQL

La cláusula GROUP BY agrupa los objetos filtrados por WHERE, a partir de un atributo de la clase, haciéndolos inaccesibles para el SELECT. En vez de ello, el SELECT podrá acceder a propiedades derivadas del grupo y utilizar las siguientes funciones de agregación:

- COUNT: el número de elementos del grupo
- SUM: el resultado de sumar los valores numéricos del grupo
- AVG: la media aritmética de los valores numéricos del grupo
- MIN: el valor numérico mínimo del grupo
- MAX: el valor numérico máximo del grupo



En el siguiente ejemplo se utiliza el GROUP BY para agrupar todas las personas por su edad y con COUNT obtener cuantas hay. En este caso se utiliza Object[] para obtener los dos parámetros de respuesta, guardando la edad en la posición 0 y la el total en la 1.

```
TypedQuery<Object[]> query
    = em.createQuery("SELECT p1.edad, COUNT(p1) FROM Persona p1 GROUP BY p1.edad",
        Object[].class);
List<Object[]> edades = query.getResultList();
for (Object[] e : edades) {
    System.out.println("edad:" + e[0] + " total:" + e[1]);
}
```

Consulta que utiliza el GROUP BY con múltiples objetos de respuesta agrupados en Object[].

En el siguiente ejemplo en vez de utilizar Object[] , tal como se explica en 15.7.1 se utiliza una estructura propia para obtener la respuesta. Para tal propósito se define la clase Edades como podemos ver en el siguiente código:

```
package ejpack;
public class Edades {
    int edad;
    long total;
    public Edades(int edad, long total) {
        this.edad = edad;
        this.total = total;
    }
}
```

Clase Edades creada específicamente para estructurar las respuestas de la consulta.



Y utilizando Edades obtenemos y mostramos los objetos de respuesta tal y como se muestra en el siguiente código:

```
TypedQuery<Edades> query
    = em.createQuery("SELECT NEW ejpack.Edades(p1.edad, COUNT(p1)) FROM Persona p1
GROUP BY p1.edad",
        Edades.class);
List<Edades> edades = query.getResultList();
for (Edades e : edades) {
    System.out.println("edad:" + e.edad + " total:" + e.total);
}
```

Consulta que utiliza el GROUP BY con múltiples objetos de respuesta agrupados en Edades[].

15.7.6. Cláusula ORDER BY en JPQL

Una vez obtenidos los contenidos a mostrar, podemos ordenarlos según uno de sus atributos de forma ascendiente o bien descendiente. Para ordenar el resultado de una consulta, añadimos al final de la consulta la cláusula ORDER BY seguido del atributo que utilizaremos para ordenar y añadiendo opcionalmente la clave DESC para ordenar descendientemente o ASC para ordenar ascendientemente. Por defecto, se ordena de forma ascendiente.

En el siguiente ejemplo obtenemos las personas según su edad descendientemente.

```
TypedQuery<Persona> query
    = em.createQuery("SELECT p1 FROM Persona p1 ORDER BY p1.edad DESC",
        Persona.class);
List<Persona> personas = query.getResultList();
for (Persona p : personas) {
    System.out.println(p);
}
```

Consulta que utiliza el ORDER BY de forma descendiente.



15.8. Modificar datos de la base de datos

Para actualizar un objeto de la base de datos primero tenemos que obtener el objeto a través de una consulta. Una vez lo tengamos, para eliminar el objeto debemos iniciar una transacción y borrar el objeto con la función *remove()* que ofrece EntityManager. Los cambios se efectuarán cuando termine la transacción con un commit.

En el siguiente ejemplo actualizamos la edad de todas las personas a 22 años.

```
TypedQuery<Persona> query = em.createQuery("SELECT p FROM Persona p", Persona.class);  
List<Persona> personas = query.getResultList();  
em.getTransaction().begin();  
    for (Persona p : personas) {  
        p.setEdad(22);  
    }  
em.getTransaction().commit();
```

Código que establece la edad de todas las personas a 22.



15.9. Borrar datos de la base de datos

Para eliminar un objeto de la base de datos, primero tenemos que obtener el objeto que queramos eliminar. Una vez lo tengamos, iniciamos una transacción y borramos el objeto con el método `remove()` de la `EntityManager`.

En el siguiente ejemplo borramos las personas con una edad de 20.

```
int e = 20;
TypedQuery<Persona> query = em.createQuery("SELECT p FROM Persona p WHERE edad=:e", Persona.class);
query.setParameter("edad", edad);
List<Persona> personas = query.getResultList();
em.getTransaction().begin();
    for (Persona p: personas) {
        System.out.println("Borrar"+p);
        em.remove(p);
    }
em.getTransaction().commit();
```

Código que muestra cómo borrar las personas con edad igual a 20.



Recursos y enlaces

- Página oficial de ObjectDB: <https://www.objectdb.com>



- Descarga ObjectDB desde: <https://www.objectdb.com/download>



- Documentación sobre JPA: <https://www.objectdb.com/java/jpa/query/jpql/structure>



Conceptos clave

- **Bases de datos orientadas a objetos:** se caracterizan porque no trabajan con los tipos de datos típicos que usan las bases de datos relacionales, sino que lo hacen con clases y objetos. Se pueden guardar y recuperar directamente los objetos manteniendo su estructura.
- **JPA:** es la API que ofrece JAVA para gestionar información relacional.
- **JDO:** es la especificación que indica como gestionar la persistencia de datos en JAVA.
- **ObjectDB:** base de datos orientada a objetos para JAVA.
- **JPQL:** lenguaje basado en una sintaxis SQL que opera con objetos.
- **JPA Criteria:** lenguaje basado en una sintaxis Java para operar con objetos.



Test de autoevaluación

¿Con qué clase gestionamos una conexión con una base de datos ObjectDB?

- a) EntityManagerFactory
- b) ObjectDBConnection
- c) EntityManager
- d) Connection

¿Qué tengo que hacer para modificar un objeto de una base de datos ObjectDB?

- e) Recuperarlo, modificarlo, y guardarlo en la base de datos con el método *store*.
- f) Recuperarlo, modificarlo, y guardarlo en la base de datos con el método *persist*.
- g) Recuperarlo de la base de datos, iniciar una transacción, modificarlo y finalizar la transacción.
- h) Recuperarlo de la base de datos, iniciar una transacción, modificarlo, guardarlo con el método *persist* y finalizar la transacción.

¿Qué clase permite guardar los múltiples resultados en una consulta múltiple?

- i) Object[]
- j) Array
- k) List[]
- l) Constraint



Ponlo en práctica

Actividad 1

Crea una clase Grupo que gestione los siguientes datos de grupos de música: nombre, estilo y número de integrantes. Crea el método constructor y los métodos get y set de cada atributo, y un método toString que muestre los datos del grupo por pantalla. La clase Grupo ha de tener un atributo id único y autogenerado

Crea un programa con el método main que utilice la clase Grupo. Se debe crear una base de datos "Grupos" en ObjectDB. El programa debe ofrecer las siguientes funcionalidades:

- Crear nuevos grupos de música y almacenarlos en la base de datos.
- Mostrar todos los grupos de la base de datos.
- Mostrar todos los grupos de un estilo concreto que indicará el usuario, ordenados alfabéticamente por su nombre.
- Modificar los datos (excepto el id) de un grupo

Actividad 2

Crea una clase Grupo que gestione los siguientes datos de grupos de música: nombre, estilo, lista de integrantes. Crea la clase Integrante que gestione el nombre y año de nacimiento del integrante. De ambas clases crea sus constructores, los métodos get y set de cada atributo, y un método toString que muestre los datos por pantalla. Ambas clases deben tener un atributo id único y autogenerado.

Crea un programa con el método main que utilice la clase Grupo e Integrante. Se debe crear una base de datos "Grupos" en ObjectDB. El programa debe ofrecer las siguientes funcionalidades:

- Crear nuevos grupos de música y almacenarlos en la base de datos.
- Mostrar todos los grupos de la base de datos.
- Mostrar todos los grupos de un estilo concreto que indicará el usuario, ordenados alfabéticamente por su nombre.
- Añadir un integrante a un grupo
- Ver los integrantes de un grupo
- Modificar los datos de un integrante
- Eliminar un integrante



SOLUCIONARIOS

Test de autoevaluación

¿Con qué clase gestionamos una conexión con una base de datos ObjectDB?

- a) EntityManagerFactory
- b) ObjectDBConnection
- c) **EntityManager**
- d) Connection

¿Qué tengo que hacer para modificar un objeto de una base de datos ObjectDB?

- a) Recuperarlo, modificarlo, y guardarlo en la base de datos con el método *store*.
- b) Recuperarlo, modificarlo, y guardarlo en la base de datos con el método *persist*.
- c) **Recuperarlo de la base de datos, iniciar una transacción, modificarlo y finalizar la transacción.**
- d) Recuperarlo de la base de datos, iniciar una transacción, modificarlo, guardarlo con el método *persist* y finalizar la transacción.

¿Qué clase permite guardar los múltiples resultados en una consulta múltiple?

- a) **Object[]**
- b) Array
- c) List[]
- d) Constraint



Ponlo en práctica

Actividad 1

Crea una clase Grupo que gestione los siguientes datos de grupos de música: nombre, estilo y número de integrantes. Crea el método constructor y los métodos get y set de cada atributo, y un método toString que muestre los datos del grupo por pantalla. La clase Grupo ha de tener un atributo id único y autogenerado.

Crea un programa con el método main que utilice la clase Grupo. Se debe crear una base de datos “Grupos” en ObjectDB. El programa debe ofrecer las siguientes funcionalidades:

- Crear nuevos grupos de música y almacenarlos en la base de datos.
- Mostrar todos los grupos de la base de datos.
- Mostrar todos los grupos de un estilo concreto que indicará el usuario, ordenados alfabéticamente por su nombre.
- Modificar los datos (excepto el id) de un grupo

El solucionario está disponible en la versión interactiva del aula.



Actividad 2

Crea una clase Grupo que gestione los siguientes datos de grupos de música: nombre, estilo, lista de integrantes. Crea la clase Integrante que gestione el nombre y año de nacimiento del integrante. De ambas clases crea sus constructores, los métodos get y set de cada atributo, y un método toString que muestre los datos por pantalla. Ambas clases deben tener un atributo id único y autogenerado.

Crea un programa con el método main que utilice la clase Grupo e Integrante. Se debe crear una base de datos "Grupos" en ObjectDB. El programa debe ofrecer las siguientes funcionalidades:

- Crear nuevos grupos de música y almacenarlos en la base de datos.
- Mostrar todos los grupos de la base de datos.
- Mostrar todos los grupos de un estilo concreto que indicará el usuario, ordenados alfabéticamente por su nombre.
- Añadir un integrante a un grupo
- Ver los integrantes de un grupo
- Modificar los datos de un integrante
- Eliminar un integrante

El solucionario está disponible en la versión interactiva del aula.