SESIÓN 1

Se comentará (recordará) en clase qué son las bases de datos orientadas a objetos y las razones de su uso para después pasar a realizar los siguientes ejercicios:

Ejercicio 1.- De la misma manera que Codd, creador del modelo relacional, definió en 1985 12 reglas que debían cumplir un sistema gestor de bases de datos para ser considerado relacional, en los años 90 se definió un conjunto de reglas equivalente para los sistemas orientados a objetos, denominado *Manifiesto de las BDOO*, en diferentes etapas y por diferentes expertos en bases de datos como el profesor Malcolm Atkinson. Dicho manifiesto estaba formado por 13 reglas obligatorias + 5 opciones de implementación no obligatorias.

Averigua y explica cada una de estas 13 reglas obligatorias (p.ej. en un fichero Word).

1. **Objetos complejos:** deben permitir construir objetos complejos aplicando constructores sobre objetos básicos.
2. **Identidad de los objetos:** todos los objetos deben tener un identificador que sea independiente de los valores de sus atributos
3. **Encapsulación:** los programadores sólo tendrán acceso a la interfaz de los métodos, de modo que sus datos e implementación estén ocultos.
4. **Tipos o clases:** el esquema de una BDOO incluye un conjunto de clases o un conjunto de tipos.
5. **Herencia:** un subtipo o una subclase heredará los atributos y métodos de su supertipo o superclase, respectivamente.
6. **Ligadura dinámica:** los métodos deben poder aplicarse a diferentes tipos (sobrecarga). La implementación de un método dependerá del tipo de objeto al que se aplique. Para proporcionar esta funcionalidad, el sistema deberá asociar los métodos en tiempo de ejecución.
7. **Su DML debe ser completo:** DML hace referencia al Lenguaje de modificación de datos
8. **El conjunto de tipos de datos debe ser extensible:** Además, no habrá distinción en el uso de tipos definidos por el sistema y tipos definidos por el usuario.
9. **Persistencia de datos:** los datos deben mantenerse después de que la aplicación que los creo haya finalizado. El usuario no tiene que hacer ningún movimiento o copia de datos explícita para ello.
10. **Debe ser capaz de manejar grandes BD:** debe disponer de mecanismos transparentes al usuario, que proporcionen independencia entre los niveles lógico y físico del sistema.
11. **Concurrencia:** debe poseer un mecanismo de control de concurrencia similar al de los sistemas convencionales
12. **Recuperación:** debe poseer un mecanismo de recuperación ante fallos similar al de los sistemas convencionales
13. **Método de consulta sencillo:** debe poseer un sistema de consulta ad-hoc de alto nivel, eficiente e independiente de la aplicación.

Ejercicio 2.- Anota las ventajas e inconvenientes (en el fichero Word) de la utilización de bases de datos orientadas a objetos.

Las ventajas de un SGBDOO son:

- **Mayor capacidad de modelado:** Un objeto permite encapsular tanto un estado como un comportamiento. Un objeto puede almacenar todas las relaciones que tenga con otros objetos. Los objetos pueden agruparse para formar objetos complejos (herencia).

- **Ampliabilidad:** Se pueden construir nuevos tipos de datos a partir de los ya existentes Agrupar propiedades comunes de diversas clases e incluirlas en una superclase, lo que reduce la redundancia. Reusabilidad de clases, lo que repercute en una mayor facilidad de mantenimiento y un menor tiempo de desarrollo.

- **Lenguaje de consulta más expresivo:** El acceso navegacional desde un objeto al siguiente es la forma más común de acceso a datos en un SGBDOO. Mientras que SQL utiliza el acceso asociativo. El acceso navegacional es más adecuado para gestionar operaciones como los despieces, consultas recursivas, etc.

- **Adecuación a las aplicaciones avanzadas de base de datos:** Hay muchas áreas en las que los SGBD tradicionales no han tenido excesivo éxito como el CAD, CASE, OIS, sistemas multimedia, etc. en los que las capacidades de modelado de los SGBDOO han hecho que esos sistemas sí resulten efectivos para este tipo de aplicaciones.

- **Mayores prestaciones:** Los SGBDOO proporcionan mejoras significativas de rendimiento con respecto a los SGBD relacionales.

Los inconvenientes de un SGBDOO son:

- **Carencia de un modelo de datos universal:** No hay ningún modelo de datos que esté universalmente aceptado para los SGBDOO y la mayoría de los modelos carecen una base teórica.

- **Carencia de experiencia:** Todavía no se dispone del nivel de experiencia del que se dispone para los sistemas tradicionales.

- **Carencia de estándares:** Existe una carencia de estándares general para los SGBDOO. - Competencia. Con respecto a los SGBDR y los SGBDOR. Estos productos tienen una experiencia de uso considerable. SQL es un estándar aprobado y ODBC es un estándar de facto. Además, el modelo relacional tiene una sólida base teórica y los productos relacionales disponen de muchas herramientas de soporte que sirven tanto para desarrolladores como para usuarios finales.

- **La optimización de consultas compromete la encapsulación:** La optimización de consultas requiere una compresión de la implementación de los objetos, para poder acceder a la base de datos de manera eficiente. Sin embargo, esto compromete el concepto de encapsulación. Bases de Datos Orientadas a Objetos José Piqueres Torres 5
- El modelo de objetos aún no tiene una teoría matemática coherente que le sirva

de base.

Ejercicio 3.- ¿Qué es el ODMG? ¿ Cuáles son los principales componentes de ODMG 3.0? (puedes responder en el fichero Word anterior).

ODMG (Object Database Mangement Group) es el grupo de fabricantes de SGBDOO que propuso el estándar ODM-93 en 1993; en 1997 evolucionó a ODMG-2.0 y en enero de 2000 se publicó la última versión ODMG 3.0.

El uso del estándar proporciona portabilidad (que se pueda ejecutar sobre sistemas distintos), interoperabilidad (que la aplicación pueda acceder a varios sistemas diferentes) y además permite que los usuarios puedan comparar entre distintos sistemas comerciales.

Los componentes principales de ODMG 3.0:

- Modelo de objetos
- Leguaje de definición de objetos (ODL)
- Lenguaje de consulta de objetos (OQL)
- Conexión con los lenguajes C++, Smaltalk y Java

Ejercicio 4.- A lo largo de esta unidad utilizaremos el SGBDOO (ODBMS) *ObjectDB* (https://www.objectdb.com/). Investiga otros 3 SGBDOO distintos indicando si siguen vigentes en la actualidad o corresponden a proyectos que están en desuso (puedes responder en el fichero Word anterior).

- **GemStone/S:** Todavía sigue vigente el proyecto, esta es la web https://gemtalksystems.com/products/gss32/
- **Versant:** Es un proyecto ya abandonado.
- **ObjectStore:** Es un proyecto ya abandonado.

SESIÓN 2

Como gestor de base de datos orientado a objetos utilizaremos ObjectDB (https://www.objectdb.com/) por tratarse de un gestor utilizado **actual**mente, rápido, seguro y fácil de usar.

Empezaremos adaptando el ejemplo que aparece en https://www.objectdb.com/tutorial/jpa/eclipse.

Para ello: Arranca el editor Eclipse. Elige del menú File > New > Java Project. En el recuadro de texto "Project name" escribe el nombre del proyecto *TutorialObjectDB* en este caso:

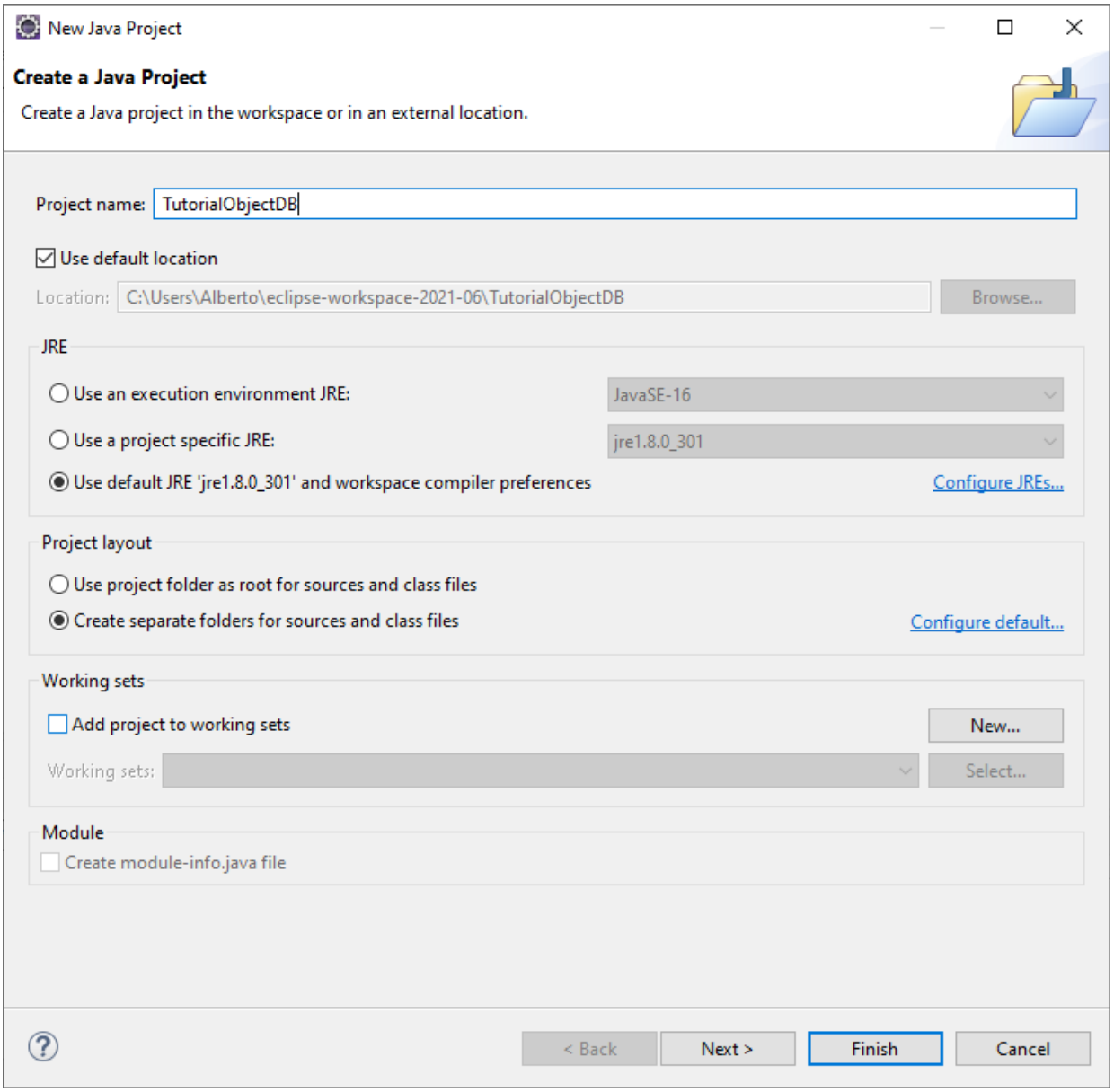


Ilustración 1

Pulsa el botón *Finish*.

Crearemos dos carpetas dentro del proyecto, la primera para guardar la librería de ObjectDB y la segunda para guardar el fichero de la base de datos.

En la pestaña [Package Explorer], pulsa botón derecho sobre el proyecto que acabamos de crear (*TutorialObjectDB*) y > New > Folder, escribiendo "librería" en el recuadro de texto "Folder Name". Seguidamente botón *Finish*.

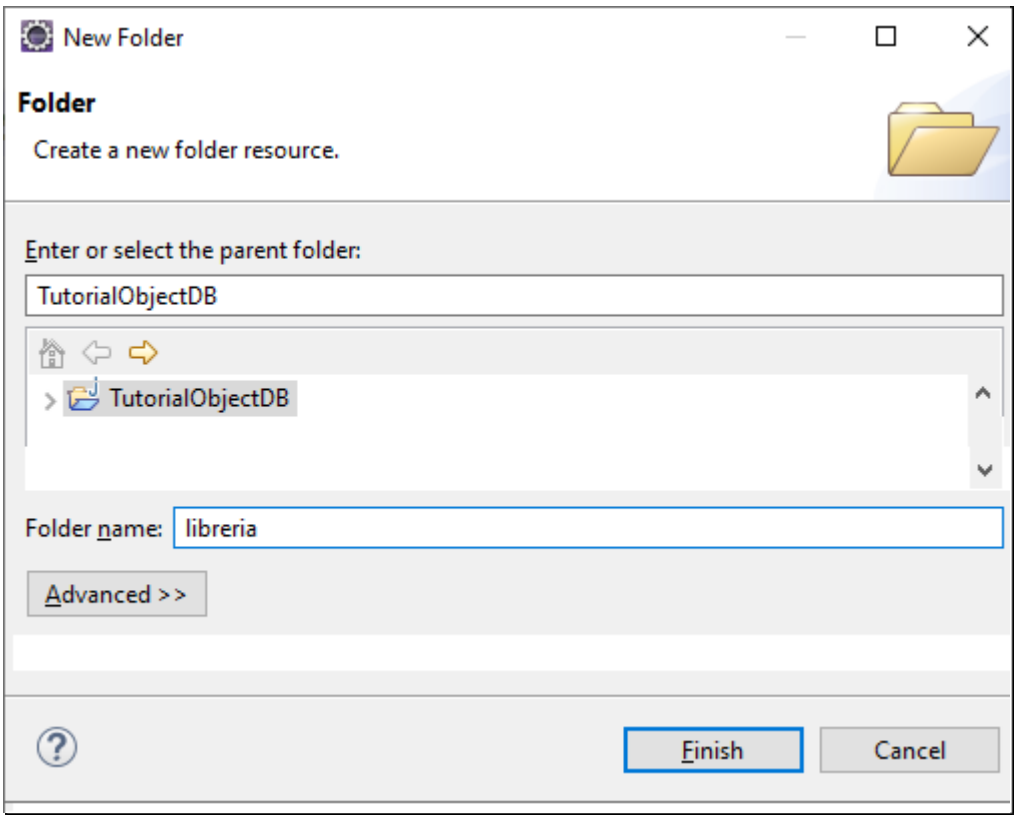


Ilustración 2

Seguimos los mismos pasos que en la ilustración 2 anterior pero para crear ahora la carpeta "db" y después de hacerlo nos quedará la siguiente estructura:

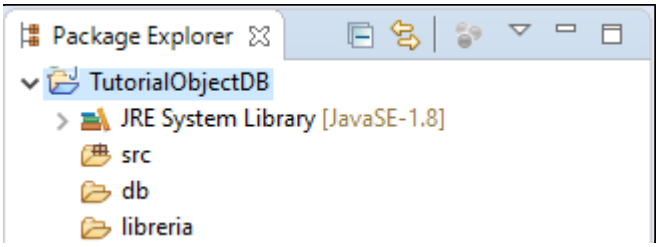


Ilustración 3

Copiamos sobre la carpeta *libreria* el fichero **objectdb.jar** del subdirectorio *bin* creado al descomprimir el **ObjectDB Development Kit 2.8.6** (fichero *objectdb-2.8.6.zip* – 6,01 MB) descargado el 5 de diciembre de 2021 de https://www.objectdb.com/download y que

corresponde a la versión de 21 de mayo de 2021. Esta carpeta y el resto de archivos los podrás encontrar en la carpeta **ficheros.**

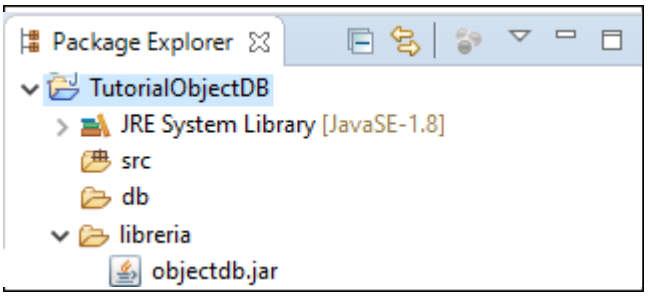


Ilustración 4

Añadiremos al proyecto la librería anterior. Para ello pulsa el botón derecho sobre el nombre del proyecto > Properties > Java Build Path > Pestaña Libraries > Botón Add JARs...

Seguidamente elegir la de la ilustración 5 y botón *OK*.

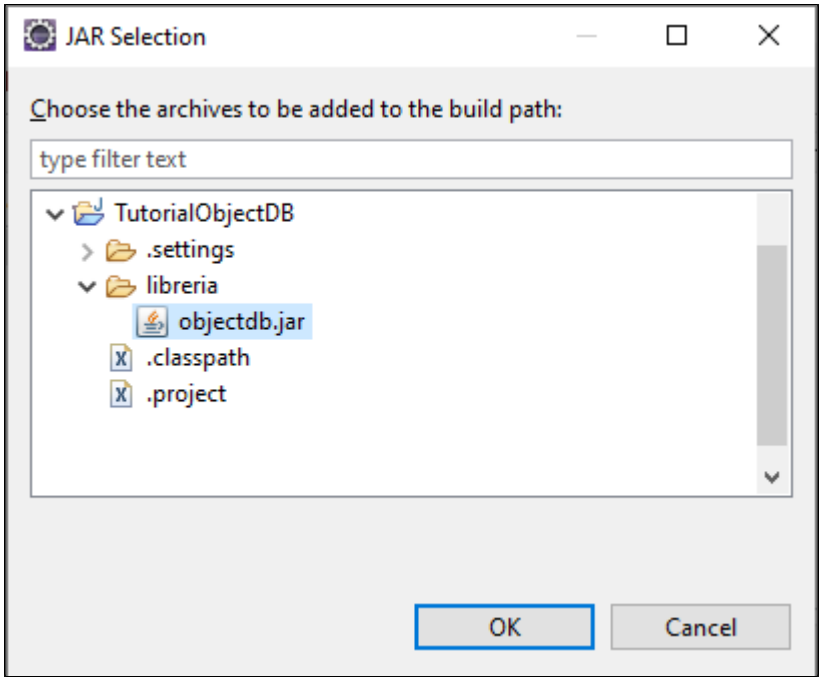


Ilustración 5

Para terminar con el botón *Apply and Close* de la ventana a la que regresamos tras la ilustración anterior.

A continuación crearemos el **paquete tutorial** dentro del proyecto *TutorialObjectDB* (botón derecho sobre el proyecto > New > Package y en recuadro de texto “Name” escribir “tutorial)

Seguidamente copiaremos sobre el paquete anterior el fichero *Point.java* que encontrarás en tu carpeta *ficheros,* dentro de la carpeta *clasesPuntos*. Es el que aparece en la siguiente ilustración.

*Point.java

```java
package tutorial;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Point implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private long id;

    private int x;
    private int y;

    public Point() {
    }

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Long getId() {
        return id;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override
    public String toString() {
        return String.format("(%d, %d)", this.x, this.y);
    }
}
```

Ilustración 6

Esta nueva clase se utilizará para representar objetos Punto en la base de datos. Se trata de una clase normal Java en la que hemos incluido anotaciones (@Entity…).

Con la anotación de la línea 6 (`@Entity`) estamos indicando que los objetos de esta clase serán persistentes, se guardarán en disco de manera permanente, a diferencia de otros objetos que se encuentran en memoria y desaparecen al terminar la aplicación.

Con las anotaciones de las líneas 10 y 11 (`@Id` y `@GeneratedValue`) estamos indicando que el atributo id de la línea 12 es una clave primaria, y será generada de manera automática (valores 1, 2, 3….) cada vez que se cree un objeto punto.

A continuación, copia sobre el paquete tutorial del proyecto los ficheros *Main00.java* y *Main01.java* de tu carpeta *ficheros*. En la ilustración 7 siguiente se comentan las operaciones que realiza esta primera clase:

*Main00.java

```java
package tutorial;
//https://www.objectdb.com/tutorial/jpa/eclipse/store
// Adaptado por Alberto Carrera Martín - Abril 2020
import javax.persistence.*;
import java.util.*;

public class Main00 {
    public static void main(String[] args) {
        // Abre una conexión a la base de datos
        // Si no existe la base de datos entonces la crea
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("db/p2.odb");
        EntityManager em = emf.createEntityManager();

        // Almacena 1.000 objetos punto en la base de datos:
        em.getTransaction().begin();
        for (int i = 0; i < 1000; i++) {
            Point p = new Point(i, i);
            em.persist(p);
        }
        em.getTransaction().commit();

        // Encuentra el número de objetos Punto en la base de datos:
        Query q1 = em.createQuery("SELECT COUNT(p) FROM Point p");
        System.out.println("Total Puntos: " + q1.getSingleResult());

        // Calcula la media del atributo X de todos los puntos:
        Query q2 = em.createQuery("SELECT AVG(p.x) FROM Point p");
        System.out.println("Media de  X: " + q2.getSingleResult());

        // Recupera todos los objetos Punto de la base de datos
        // y los almacena en una lista (línea 35)
        // Después recorre todos los puntos de esa lista (línea 36)
        TypedQuery<Point> query =
            em.createQuery("SELECT p FROM Point p", Point.class);
        List<Point> results = query.getResultList();
        for (Point p : results) {
            System.out.println(p);
        }

        // Cierra la consulta y la conexión a la base de datos
        em.close();
        emf.close();
    }
}
```

Ilustración 7

Ejecutamos la aplicación de la ilustración 7 anterior haciendo clic sobre ella y comando del menú Run > Run y vemos el resultado en la ventana Console (ilustración 8).

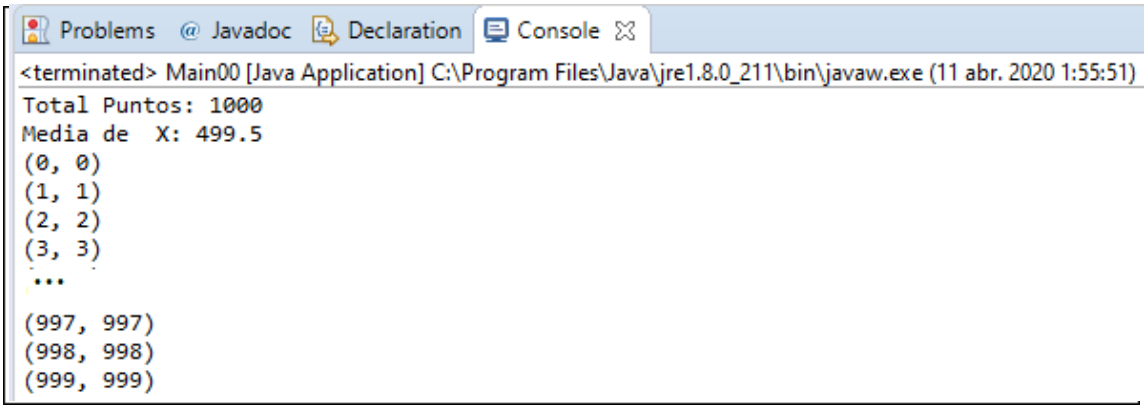


Ilustración 8

Si refrescas la carpeta *db* del proyecto (Botón derecho sobre ella > Refresh)

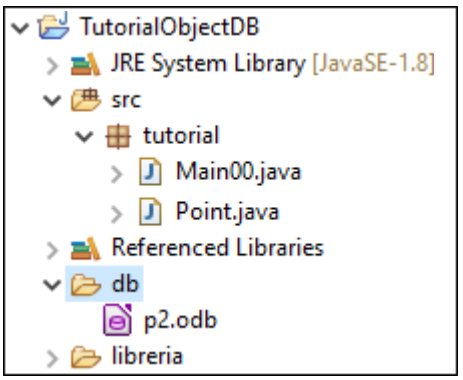


Ilustración 9

comprobarás que se ha creado el fichero de la base de datos, *p2.odb*.

Si volvieras a ejecutar la aplicación de la ilustración 9, como ya existe la base de datos, volverías a crear 1.000 puntos más y pasarías a tener 2.000 y así sucesivamente.

Podrías borrar la base de datos en cualquier momento pulsando el botón derecho del ratón sobre ella > Delete > Ok. Recuerda que cuando intentamos abrir una conexión sobre una base de datos y ésta no existe, se crea nueva.

Haremos lo mismo que hemos realizado desde la ilustración 8 pero para la segunda clase "Main" (Main01, cuyo fichero *Main01.java* aparece en la carpeta *ficheros*). En la ilustración 10 siguiente se comentan las operaciones que realiza esta clase y en la posterior ilustración el resultado de su ejecución:

*Main01.java

```
1  package tutorial;
2  //https://www.objectdb.com/tutorial/jpa/eclipse/store
3  //Adaptado por Alberto Carrera Martín - Abril 2020
4  import javax.persistence.*;
5  import java.util.*;
6
7  public class Main01 {
8      public static void main(String[] args) {
9          // Abre una conexión a la base de datos
10         // Si no existe la base de datos entonces la crea
11         EntityManagerFactory emf =
12             Persistence.createEntityManagerFactory("db/p2.odb");
13         EntityManager em = emf.createEntityManager();
14
15      // Recupera todos los objetos Punto de la base de datos:
16         TypedQuery<Point> query =
17                 em.createQuery("SELECT p FROM Point p", Point.class);
18               List<Point> results = query.getResultList();
19               // Borra todos los puntos menos los 100 primeros
20               // El resto incrementa su coordenada X en 100 unidades
21               em.getTransaction().begin();
22               for (Point p : results) {
23                   if (p.getX() >= 100) {
24                       em.remove(p); // Borra la entidad
25                   }
26                   else {
27                       p.setX(p.getX() + 100); // Modifica la entidad
28                   }
29               }
30               em.getTransaction().commit();
31
32         // Recupera todos los objetos Punto que permanecen:
33         query = em.createQuery("SELECT p FROM Point p", Point.class);
34         results = query.getResultList();
```

Ilustración 10

Problems @ Javadoc Declaration Console

```
<terminated> Main01 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (11 abr. 2020 2:16:12)
(100, 0)
(101, 1)
(102, 2)
...
(197, 97)
(198, 98)
(199, 99)
```

Si quieres ver de otra manera el contenido del fichero *p2.odb* lo puedes hacer abriendo la herramienta **ObjectDB Explorer** que se encuentra en la carpeta *bin* (ilustración 12).

ObjectDB Database Explorer es una herramienta visual GUI para administrar bases de datos ObjectDB. Se puede usar para ver datos en bases de datos ObjectDB, ejecutar distintos tipos de consultas y editar el contenido de las bases de datos.

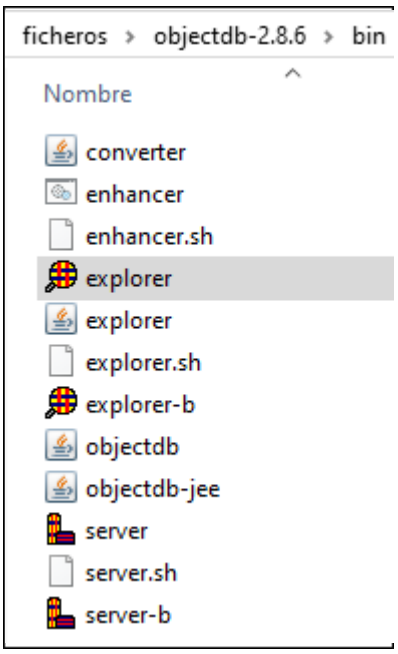
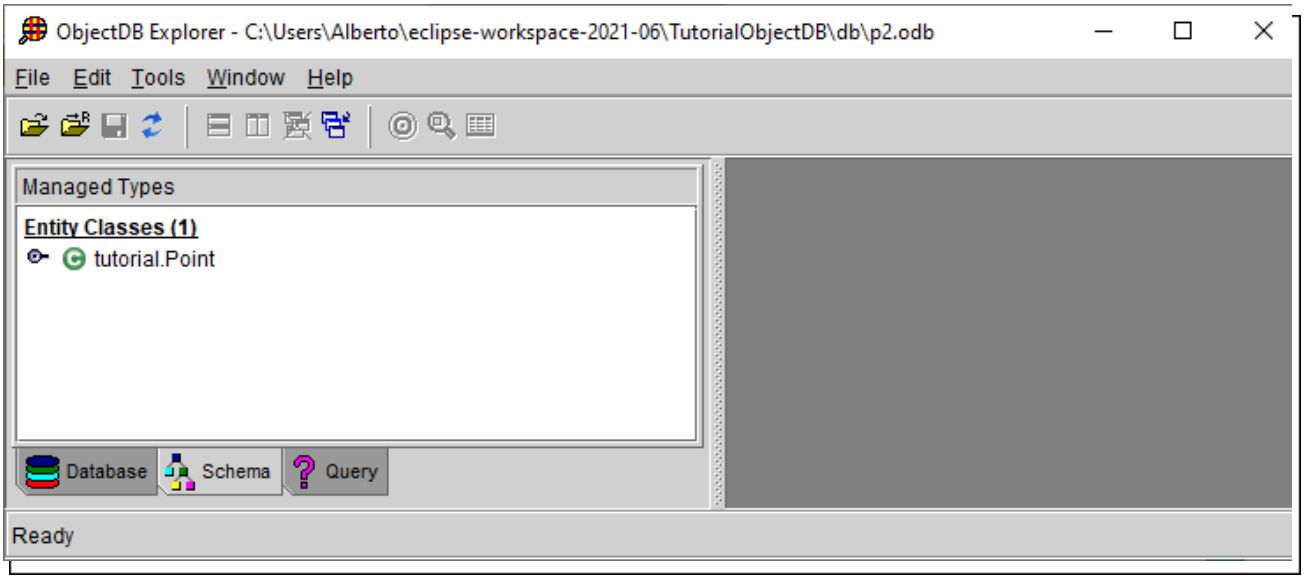


Ilustración 12

Al abrirla (doble clic sobre ella) aparece el último fichero de base de datos utilizado. Si no apareciera o en cualquier momento quieres cambiar de base de datos entonces utiliza la opción *Open Embedded ...* del menú *File*.



Seleccionando con un clic la clase Point y eligiendo del menú el tipo de vista (*Window/Open Tree Window* ó *Window/Open Table Window*), veremos el contenido de la base de datos. En el caso de la imagen siguiente los 100 objetos punto se visualizan en forma de tabla(opción Window/Open Table Window), representando cada fila un objeto punto y cada columna los atributos del mismo. Observa además que la primera columna, id, es el atributo clave primaria que hemos declarado en las líneas 10 a 12 de la ilustración 6.

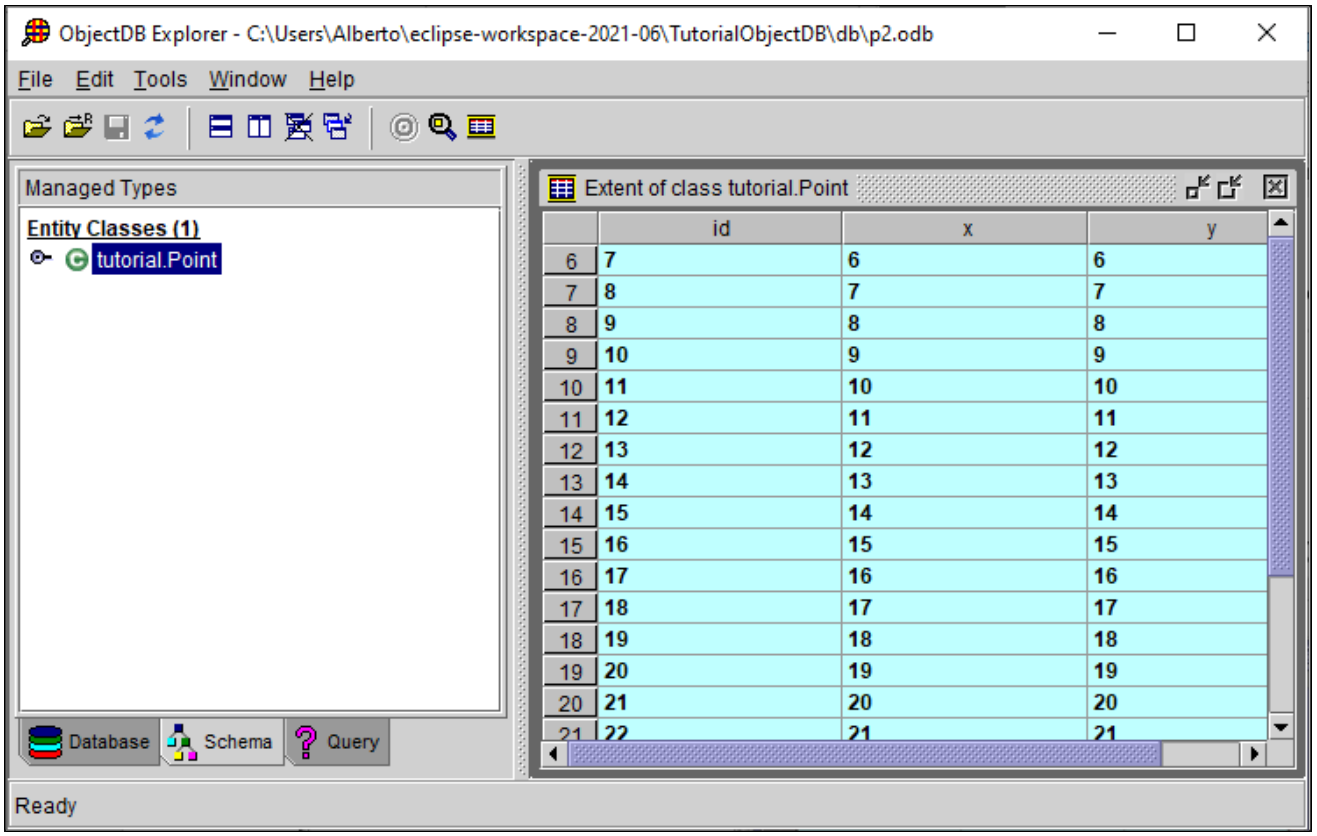


Ilustración 14

Muy interesante la pestaña **? Query** (parte inferior izquierda de la ilustración) para lanzar y probar todas las consultas que necesites (más tarde conocerás el lenguaje para poder crearlas y utilizarlas)

Con esta introducción finalizaría una pequeña demostración de uso de la base de datos orientada a objetos ObjectDB. A continuación pasaremos a explicar el funcionamiento de una aplicación CRUD con un poco más de detalle. Podrás encontrar las clases dentro de la carpeta **ficheros**. Bastará por tanto que crees un proyecto y añadas la librería y las clases de la carpeta anterior.

Ejercicio 5 .- Dedica un tiempo a probar la herramienta **ObjectDB Explorer** anterior.

Ejercicio 6 .- ¿Con qué otras dos interfaces trabajadas con la herramienta Hibernate relacionarías las interfaces EntityManagerFactory y EntityManager? (anota la respuesta en el fichero Word).

Con las interfaces **SessionFactory** y **Session**

Ejercicio 7.- En los ejemplos anteriores se han utilizado en las consultas dos interfaces distintas, Query y TypedQuery. ¿En qué se diferencian? (averigualo anota la respuesta en el fichero Word).

Query: Se implementó en la versión JPA1 y se utiliza cuando desconocemos el tipo de resultado de la consulta o es un resultado polimórfico.

TypedQuery: Se implementó en la versión JPA2 y se utiliza cuando conocemos el tipo de resultado de la consulta, con esto es más fácil procesar los resultados.

SESIÓN 3

Para la siguiente práctica puedes utilizar un proyecto Java nuevo y dentro de él crear el paquete *ejemplo1* y copiar las clases que encontrarás en la carpeta *clasesEmpleadosDepartamentos* incluida dentro de la carpeta *ficheros*. Dichas clases se explican a continuación

CLASE DepartamentoEntity.java

DepartamentoEntity.java

```
1  package ejemplo1;
2
3  import java.util.HashSet;
9  //
10 //Alberto Carrera Martín - Abril 2020
11 //
12 @Entity
13 public class DepartamentoEntity {
14     @Id
15     private int dptoId;
16     private String nombre;
17     private String localidad;
18     @OneToMany(mappedBy="departamento")
19         private Set<EmpleadoEntity> empleados = new HashSet<EmpleadoEntity>();
20     //
21     public DepartamentoEntity(int dptoId, String nombre, String localidad) {
22
23         this.dptoId = dptoId;
24         this.nombre = nombre;
25         this.localidad = localidad;
26
27     }
```

Ilustración 15

Se trata de una clase que tiene que ser persistida, es decir almacenada y conservada tras la finalización de la aplicación. Ello viene indicado con la anotación `@Entity` de la línea 12. La clave primaria de los departamentos será el atributo *dptoId* como así lo expresa la anotación `@Id` de la línea 14; no se indica que se genera automáticamente dicha clave por lo que tendremos que suministrar el valor cuando creemos cada departamento.

La única "novedad" con respecto al proyecto demo anterior que se presenta en la ilustración 15 anterior es la definición de una relación 1 a M entre Departamentos y Empleados. Esta relación se expresa con la anotación `@OneToMany` de la línea 18, indicando además que el atributo departamento de la clase Empleados (`mappedBy="departamento"`) será el que especifique esta relación. No indicamos ningún tipo de comportamiento sobre los empleados en operaciones de actualización o eliminación de departamentos. Cada departamento mantendrá una referencia al conjunto de empleados que lo forman (línea 19), en este caso utilizando la colección conjunto (`private Set<EmpleadoEntity> empleados = new HashSet<EmpleadoEntity>()`) dentro del último atributo, *empleados*, de esta clase Departamentos.

En el lado inverso de la relación, en la clase Empleados, también se tiene constancia de ésta por la referencia de la línea 24 (`@ManyToOne`) de la siguiente ilustración 16.

CLASE EmpleadoEntity.java

*EmpleadoEntity.java

```
1  package ejemplo1;
2
3  import java.util.Date;
9
10 //
11 //Alberto Carrera Martín - Abril 2020
12 //
13
14 @Entity
15 public class EmpleadoEntity {
16     @Id
17     private int empnoId;
18     private String nombre;
19     private String oficio;
20     private EmpleadoEntity dirId;
21     private Date alta;
22     private Integer salario;
23     private Integer comision;
24     @ManyToOne
25         private DepartamentoEntity departamento;
```

Ilustración 16

La clave primaria de los empleados será el primer atributo, *empnoId* (línea 17). Observad que en esta clase, dos de los atributos no son simples sino que almacenan referencias a otros objetos: El atributo *dirId* (línea 20) referencia al Jefe del empleado y el atributo *departamento* (línea 25) "apunta" al departamento al que pertenece el empleado.

CLASE MainCreacion.java

(ilustración 17 página siguiente)

Equivaldría al "script" de creación de la base de datos empleados. En las líneas 15 a 19 y 22 a 37 se crean los objetos departamentos y empleados que posteriormente se almacenarán. En las líneas 39 a 41 creamos la conexión con la base de datos, como no existe ésta se creará. Comenzamos la transacción en la línea 42 (`em.getTransaction().begin()`) y la finalizaremos en la línea 47 (`em.getTransaction().commit()`) tras haber hecho persistentes (permanentes) los 5 departamentos + los 14 empleados (`em.persist(..)`). Para finalizar cerramos la conexión en las líneas 48 y 49.

*MainCreacion.java

```java
package ejemplo1;
import java.text.ParseException;
//
//Alberto Carrera Martín - Abril 2020
//
public class MainCreacion {

    public static void main(String[] args) throws ParseException {
        DepartamentoEntity d1 = new DepartamentoEntity (10, "Finanzas", "Huesca");
        DepartamentoEntity d2 = new DepartamentoEntity (20, "I+D", "Walqa-Cuarte");
        DepartamentoEntity d3 = new DepartamentoEntity (30, "Comercial", "Almudévar");
        DepartamentoEntity d4 = new DepartamentoEntity (40, "Producción", "Barbastro");
        DepartamentoEntity d5 = new DepartamentoEntity (50, "Marketing", "Zaragoza");

        SimpleDateFormat formato = new SimpleDateFormat("yyyy-MM-dd");
        EmpleadoEntity e1 = new EmpleadoEntity (1039, "Alberto Carrera Martín", "Presidente", null, formato.parse("1999-10-27"), 4900, null, d1);
        EmpleadoEntity e2 = new EmpleadoEntity (1082, "Mario Carrera Bailín","Director", e1, formato.parse("2001-07-06"), 3385, null, d1);
        EmpleadoEntity e3 = new EmpleadoEntity (1034, "Raquel Carrera Bailín","Empleado",e2, formato.parse("2002-11-12"), 2690, null,d1);

        EmpleadoEntity e4 = new EmpleadoEntity (2066, "Blanca Bailín Perarnau", "Director", e1, formato.parse("2001-07-12"), 2970, null, d2);
        EmpleadoEntity e5 = new EmpleadoEntity (2002, "Araceli Carrera Salcedo", "Investigador", e4, formato.parse("2003-02-24"),3000 ,null, d2);
        EmpleadoEntity e6 = new EmpleadoEntity (2069, "Fernando Carrera Martín", "Empleado", e5, formato.parse("2001-11-19"),2840, null, d2);
        EmpleadoEntity e7 = new EmpleadoEntity (2088, "Carmen Bailín Perarnau", "Investigador", e4, formato.parse("2001-10-19"), 2600, null, d2);
        EmpleadoEntity e8 = new EmpleadoEntity (2076, "Fernando Carrera Salcedo", "Empleado",e7, formato.parse("2005-02-13"), 2730,null, d2);
        //
        EmpleadoEntity e9 = new EmpleadoEntity (3098, "Fernando Martínez Pérez", "Director",  e1, formato.parse("2000-02-03"), 3150,  null, d3);
        EmpleadoEntity e10 = new EmpleadoEntity (3099, "Belén Carrera Sausán", "Comercial", e9, formato.parse("2000-02-22"), 2500, 500, d3);
        EmpleadoEntity e11 = new EmpleadoEntity (3051, "Enrique Casado Alvarez", "Comercial", e9, formato.parse("2002-07-23"), 2600, 550, d3);
        EmpleadoEntity e12 = new EmpleadoEntity (3054, "Antonio Mériz Piedrafita", "Comercial", e9, formato.parse("2003-03-22"), 2600, 1000, d3);
        EmpleadoEntity e13 = new EmpleadoEntity (3044,"Lorenzo Blasco González", "Comercial", e9, formato.parse("2001-03-07"), 2350, 400 ,d3);
        EmpleadoEntity e14 = new EmpleadoEntity (3000,"Javier Escartín Nasarre", "Empleado", e9, formato.parse("2003-07-13"), 2435, null, d3);
        //
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("db/empleados.odb");
       EntityManager em = emf.createEntityManager();
       em.getTransaction().begin();
       em.persist(d1); em.persist(d2);em.persist(d3);em.persist(d4);em.persist(d5);
       em.persist(e1);em.persist(e2);em.persist(e3);em.persist(e4);em.persist(e5);em.persist(e6);em.persist(e7);
       em.persist(e8);em.persist(e9);em.persist(e10);em.persist(e11);em.persist(e12);em.persist(e13);em.persist(e14);
       //
       em.getTransaction().commit();
       em.close();
       emf.close();
    }
}
```

Ilustración 17

CLASE AccesoBdatos.java

AccesoBdatos.java

```
1  package ejemplo1;
2
3  import java.util.List;
11 //
12 // Alberto Carrera Martín - Abril 2020
13 //
14
15 public class AccesoBdatos {
16     private EntityManagerFactory emf;
17     private EntityManager em;
18
19     public void conectar() {
20         emf = Persistence.createEntityManagerFactory("db/empleados.odb");
21         em = emf.createEntityManager();
22     }
23     public void desconectar() {
24         em.close();
25         emf.close();
26     }
```

Ilustración 18

Esta clase, como hemos hecho con el modelo relacional, es la que contiene todos los métodos de acceso, recuperación y manipulación de datos. Solamente consta de 2 atributos necesarios para la conexión (líneas 16 y 17). El atributo *em* es el que contiene realmente los datos de la conexión y sobre el que se aplican los métodos de buscar, guardar… de manera muy similar a una conexión relacional.

Utilizaremos la clase Main2.java para probar los distintos métodos de esta clase AccesoBdatos.java.

Comenzaremos por un método sencillo, `public DepartamentoEntity buscarDepartamento(int numDepartamento)` de la línea 27 de la ilustración 22. El método recibe un número de departamento como argumento y devuelve el objeto departamento correspondiente a ese número o null en caso de no existir. Se apoya en el método find que permite buscar objetos por clave primaria (https://www.objectdb.com/api/java/jpa/EntityManager/find_Class Object)

El método `public void imprimirDepartamento (int numDepartamento)` (línea 32 de la ilustración 22, se detalla más adelante su funcionamiento) recibe un número de departamento e indica todos sus datos, así como los empleados.

Ejemplos de ejecución (recuerda probarlos desde Main2.java).

Ilustración 19. `abd.imprimirDepartamento(90);`

Ilustración 60. `abd.imprimirDepartamento(40);`

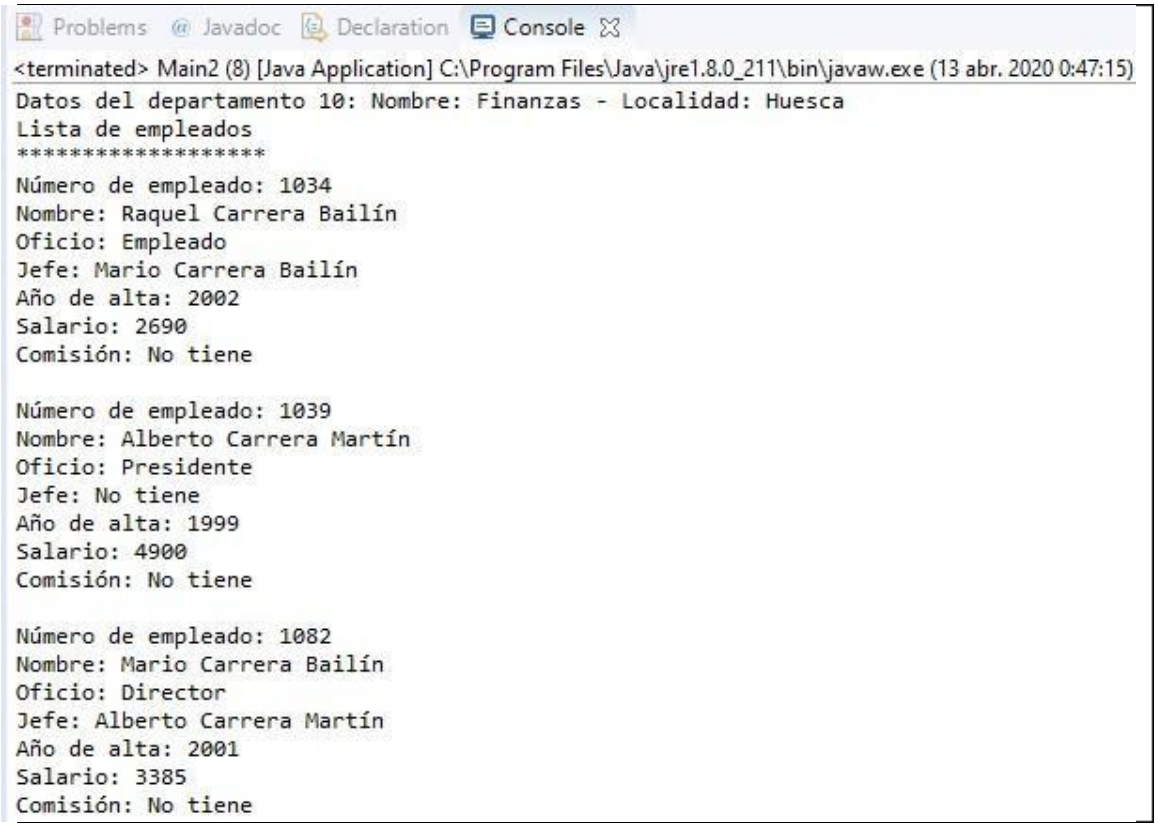


Ilustración 71. `abd.imprimirDepartamento(10);`

Comentarios al método `public void imprimirDepartamento (int numDepartamento)` (línea 32):

Lo primero de todo es comprobar que el departamento existe, línea 33, apoyándose en el método `public DepartamentoEntity buscarDepartamento(int numDepartamento)` comentado anteriormente. Si no lo encuentra (línea 34) avisa de ello por la ventana de cónsola, en caso contrario (línea 36) se prepara para obtener sus datos así como el del conjunto de empleados que lo forman. Este **conjunto** de empleados lo obtenemos con uno de los métodos getter de la clase en la línea 37. También utilizamos otros métodos getter para conocer el nombre y la localidad. Toda la información la recogeremos en el objeto cadena *datos* de la línea 38. En las líneas 40 a 44 nos aseguramos si el departamento tiene o no empleados para utilizar un rótulo de salida u otro. Entre las líneas 46 a 60 se recorre el conjunto de empleados del departamento. Esta búsqueda podía haberse incluido en el bloque "else" anterior pero se ha dejado separada por claridad visual. De cada empleado nos vamos quedando con su número, nombre, oficio (líneas 47 a 49)… según tenga jefe o no (líneas 50 a 53) ponemos un rótulo indicando que no lo tiene o incluimos su nombre en el caso que si lo tenga. Una cosa muy parecida haríamos con la comisión… Al final en la línea 62 se imprimen todos los datos que se han ido concatenando en la cadena *datos*.

AccesoBdatos.java

```
27    public DepartamentoEntity buscarDepartamento(int numDepartamento) {
28        return em.find(DepartamentoEntity.class, numDepartamento);
29    }// de método buscarDepartamento
30    //
31    @SuppressWarnings("deprecation")
32    public void imprimirDepartamento (int numDepartamento) {
33        DepartamentoEntity d = buscarDepartamento(numDepartamento);
34        if (d==null)
35            System.out.println("No existe el Departamento " + numDepartamento);
36        else {
37            Set <EmpleadoEntity> empleados =d.getEmpleados();
38            String datos="Datos del departamento " + numDepartamento + ": ";
39            datos+= "Nombre: " + d.getNombre() + " - Localidad: " + d.getLocalidad()+ "\n";
40            if (empleados.isEmpty())
41                datos+="No tiene empleados en este momento";
42            else {
43                datos+="Lista de empleados"+ "\n";
44                datos+="******************";
45            }
46            for (EmpleadoEntity empleado :empleados) {
47                datos+= "\nNúmero de empleado: " + empleado.getEmpnoId()+ "\n";
48                datos+= "Nombre: " + empleado.getNombre()+ "\n";
49                datos+= "Oficio: " + empleado.getOficio()+ "\n";
50                if (empleado.getDirId()==null)
51                    datos+= "Jefe: No tiene"+ "\n";
52                else
53                    datos+= "Jefe: "+ empleado.getDirId().getNombre()+ "\n";
54                datos+= "Año de alta: " + (empleado.getAlta().getYear()+1900)+ "\n";
55                datos+= "Salario: "+ empleado.getSalario()+ "\n";
56                if (empleado.getComision() ==null)
57                    datos+= "Comisión: No tiene"+ "\n";
58                else
59                    datos+= "Comisión: "+ empleado.getComision()+ "\n";
60            }
61
62            System.out.println(datos);
63        }
64    } // de método imprimirDepartamento
```

Ilustración 82

El método `public boolean insertarDepartamento (DepartamentoEntity d)` de la ilustración 23 recibe como argumento el departamento a insertar en la base de datos. Se comprueba que no exista el departamento antes de insertarlo. Si existe (línea 67) el método termina devolviendo false (línea 68) como resultado de la inserción. Por otro lado, si el departamento es nuevo, lo guarda a través de la transacción de las líneas 69 a 71, devolviendo true (línea 72) para indicar el resultado de la ejecución.

AccesoBdatos.java

```
65
66      public boolean insertarDepartamento (DepartamentoEntity d) {
67          if (buscarDepartamento(d.getDptoId())!=null)
68              return false;
69          em.getTransaction().begin();
70          em.persist(d);
71          em.getTransaction().commit();
72          return true;
73      } // de insertarDepartamento
```

Ilustración 93

En la ilustración 24, probamos el método anterior intentando guardar dos veces un mismo departamento; la primera vez (línea 20) se almacena el departamento (devolviendo true en la parte inferior izquierda de la ilustración) pero la segunda vez que lo intentamos (línea 21) ya no lo guardará (false) al existir un departamento con este número. Comprobamos en la línea 22 los datos del nuevo departamento insertado:

```
20 System.out.println(abd.insertarDepartamento(new DepartamentoEntity(60,"Recursos Humanos", "Chimillas")));
21 System.out.println(abd.insertarDepartamento(new DepartamentoEntity(60,"Recursos Humanos", "Chimillas")));
22 abd.imprimirDepartamento(60);
23
```

Problems @ Javadoc Declaration Console

```
<terminated> Main2 (8) [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (13 abr. 2020 12:39:05)
true
false
Datos del departamento 60: Nombre: Recursos Humanos - Localidad: Chimillas
No tiene empleados en este momento
```

Ilustración 104

El método **`public boolean modificarDepartamento (DepartamentoEntity d)`** recibe como argumento un departamento que contiene el nombre y/o localidad nuevos a actualizar. Si el departamento no existe (líneas 76 a 78) no hay nada que actualizar y el método finaliza devolviendo false. En caso contrario, en las líneas 79 a 83 se procede a cambiar los datos del departamento (nombre y localidad) por los nuevos devolviendo true para confirmar que la operación fue un éxito.

```
75    public boolean modificarDepartamento (DepartamentoEntity d) {
76        DepartamentoEntity departamentoBuscado=buscarDepartamento(d.getDptoId());
77        if (departamentoBuscado==null)
78            return false;
79        em.getTransaction().begin();
80        departamentoBuscado.setNombre(d.getNombre());
81        departamentoBuscado.setLocalidad(d.getLocalidad());
82        em.persist (departamentoBuscado);
83        em.getTransaction().commit();
84        return true;
85    } // de modificarDepartamento
```

Ilustración 115

La comprobación del método anterior la encontramos en la siguiente ilustración 26. No hemos podido cambiar los datos del departamento 88 pues no existe (false parte inferior de la ilustración), en cambio si que hemos podido cambiar los datos del departamento 60 dejando su nombre como RRHH y trasladando su sede a Esquedas.

```
24 System.out.println(abd.modificarDepartamento(new DepartamentoEntity(88,"RRHH", "Huerrios")));
25 System.out.println(abd.modificarDepartamento(new DepartamentoEntity(60,"RRHH", "Esquedas")));
26 abd.imprimirDepartamento(60);
27
```

Problems @ Javadoc Declaration Console

```
<terminated> Main2 (8) [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (13 abr. 2020 19:07:48)
false
true
Datos del departamento 60: Nombre: RRHH - Localidad: Esquedas
No tiene empleados en este momento
```

Ilustración 26

El método **`public boolean`** `borrarDepartamento (int numDepartamento)` intenta borrar el departamento cuyo número se le pasa como argumento, siempre y cuando este exista y no tenga empleados (líneas 92 a 93):

```
90    public boolean borrarDepartamento  (int numDepartamento) {
91        DepartamentoEntity departamentoBuscado=buscarDepartamento(numDepartamento);
92        if (departamentoBuscado==null || !departamentoBuscado.getEmpleados().isEmpty() )
93            return false;
94        em.getTransaction().begin();
95        em.remove(departamentoBuscado);
96        em.getTransaction().commit();
97        return true;
98    } // de modificarDepartamento
```

Ilustración 127

En la siguiente ilustración comprobamos que solo ha dejado borrar el 60, pues existe y no tiene empleados. El departamento 88 no se puede borrar porque no existe y el 10 tampoco pues existe pero tiene empleados adscritos a él.

```
28  abd.borrarDepartamento(88); // false no existe
29  abd.borrarDepartamento(60); // true
30  abd.borrarDepartamento(10); // false pues tiene empleados
```

Ilustración 28

Terminaremos explicando por partes el último método que nos queda:

`public void demoJPQL()` en el que se han intentado incluir diferentes consultas utilizando el lenguaje de consulta JPQL que como podrás comprobar es muy parecido al SQL que conoces. Las dos consultas que se encuentran en las líneas 101-102 y 10-106 de la ilustración 29 hacen lo mismo, devuelven el total de departamentos de la base de datos (posiblemente sea 5 según las pruebas que hayas hecho por tu cuenta con los datos…)

La primera de ellas utiliza la interfaz Query y la segunda la TypedQuery.

La interfaz Query es más antigua (https://www.objectdb.com/api/java/jpa/Query) y la nueva TypedQuery (https://www.objectdb.com/api/java/jpa/TypedQuery) hereda de la anterior y es más "moderna".

La interfaz Query se usa principalmente cuando el tipo de resultado de la consulta es desconocido. Es más eficiente ejecutar consultas y procesar los resultados de la misma de forma segura cuando se usa la interfaz TypedQuery.

Observa las líneas 104 y 105 al utilizar la interfaz TypedQuery. El dato devuelto “es conocido”, Long, y por eso queda claramente especificado.

Tanto en una consulta como en otra utilizamos el método `getSingleResult()`

(https://www.objectdb.com/api/java/jpa/Query/getSingleResult) que se usa siempre que se va a recuperar un dato sencillo (en este caso el número 5 que indica el total de departamentos)

```
100    public void demoJPQL() {
101        Query q1 = em.createQuery("SELECT COUNT(d) FROM DepartamentoEntity d");
102        System.out.println("Total Departamentos: " + q1.getSingleResult());
103        //
104        TypedQuery<Long> tq1 = em.createQuery(
105                "SELECT COUNT(d) FROM DepartamentoEntity d", Long.class);
106        System.out.println("Total Departamentos: " + tq1.getSingleResult());
```

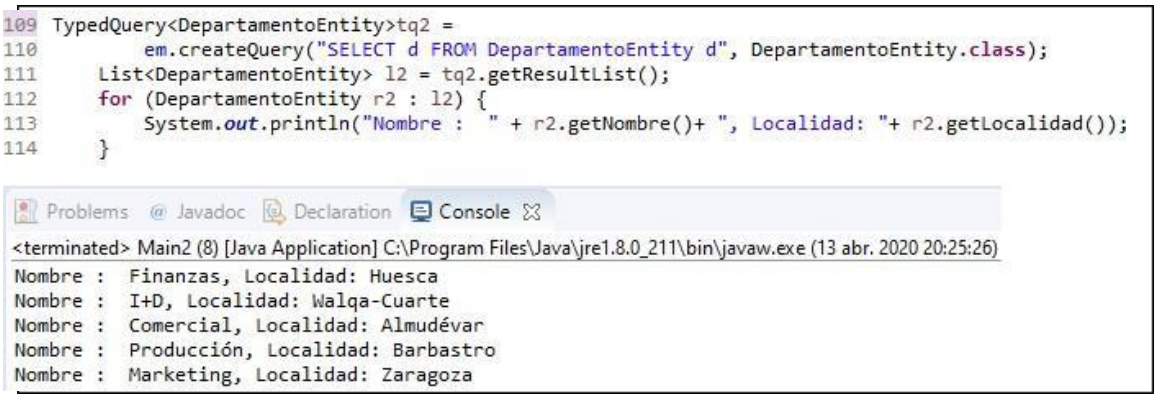
Ilustración 29

Ejecutando las dos consultas anteriores:

```
Problems  @ Javadoc  Declaration  Console
<terminated> Main2 (8) [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (13 abr. 2020 19:52:16)
Total Departamentos: 5
Total Departamentos: 5
```

Ilustración 130

La siguiente consulta recupera todos los departamentos. En la línea 110 se crea la consulta que se ejecuta en la 111 recuperando la lista de los departamentos (método `getResultList()`) y dejándolos en el objeto lista l2 que posteriormente pasamos a recorrer. Recuperamos los objetos completos aunque solo imprimimos el nombre y la localidad.



```
109 TypedQuery<DepartamentoEntity>tq2 =
110         em.createQuery("SELECT d FROM DepartamentoEntity d", DepartamentoEntity.class);
111     List<DepartamentoEntity> l2 = tq2.getResultList();
112     for (DepartamentoEntity r2 : l2) {
113         System.out.println("Nombre :  " + r2.getNombre()+ ", Localidad: "+ r2.getLocalidad());
114     }
```

Problems @ Javadoc Declaration Console

```
<terminated> Main2 (8) [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (13 abr. 2020 20:25:26)
Nombre :  Finanzas, Localidad: Huesca
Nombre :  I+D, Localidad: Walqa-Cuarte
Nombre :  Comercial, Localidad: Almudévar
Nombre :  Producción, Localidad: Barbastro
Nombre :  Marketing, Localidad: Zaragoza
```

Ilustración 141

A diferencia de la anterior, la siguiente consulta (ilustración 32) no recupera objetos enteros sino atributos “sueltos”, en este caso nombre y la localidad (línea 117). **No existe un dato predefinido de Java o definido por el usuario** como en los casos anteriores (Long.class o DepartamentoEntity.class), por eso en esta situación se tiene que recuperar la información de “cada fila” como un conjunto (vector) de objetos (recuadro rojo líneas 116-117). En la línea 118 se ejecuta la consulta dejando los resultados en una lista l3 donde cada elemento de esta lista, es a su vez es un vector de objetos. En nuestro caso la lista estaría formada por 5 elementos. Cuando se recorre el resultado a partir de la línea119, la primera iteración correspondería al primer vector de objetos de la lista l3, que se almacenará en el vector r3 que es el que se utiliza para el recorrido. La primera posición del vector r3[0] correspondería al primer atributo de la SELECT (en este caso nombre y valor Finanzas) y la segunda posición r3[1] al segundo atributo de la SELECT (localidad y valor Huesca). En la segunda iteración se dejaría el segundo resultado recuperado de la SELECT otra vez en r3, siendo r3[0] I+d y r3[1] Walqa-Cuarte… **Quizás de forma gráfica, ilustración 33, quede más claro**

```
116 TypedQuery<Object[]>tq3 =
117         em.createQuery("SELECT d.nombre, d.localidad FROM DepartamentoEntity  d", Object[].class);
118     List<Object[]> l3 = tq3.getResultList();
119     for (Object[] r3 : l3) {
120         System.out.println(
121         "Nombre :  " + r3[0] + ", Localidad: " + r3[1]);
```

Problems Javadoc Declaration Console

```
<terminated> Main2 (8) [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (13 abr. 2020 20:40:55)
Nombre :  Finanzas, Localidad: Huesca
Nombre :  I+D, Localidad: Walqa-Cuarte
Nombre :  Comercial, Localidad: Almudévar
Nombre :  Producción, Localidad: Barbastro
Nombre :  Marketing, Localidad: Zaragoza
```

Ilustración 152

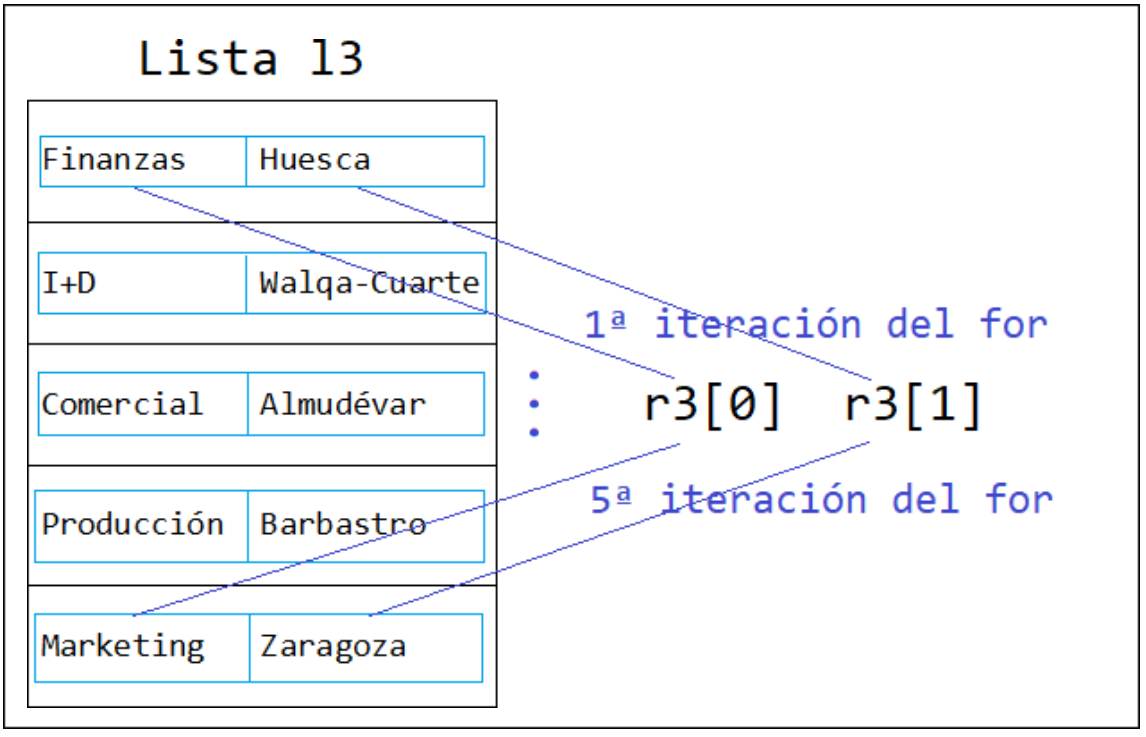


Ilustración 163

La siguiente consulta de la ilustración 34 es muy parecida a la anterior, recupera todos los departamentos menos el 10. Observa que hemos utilizado una estructura muy parecida a las sentencias preparadas con **utilización de parámetros**:

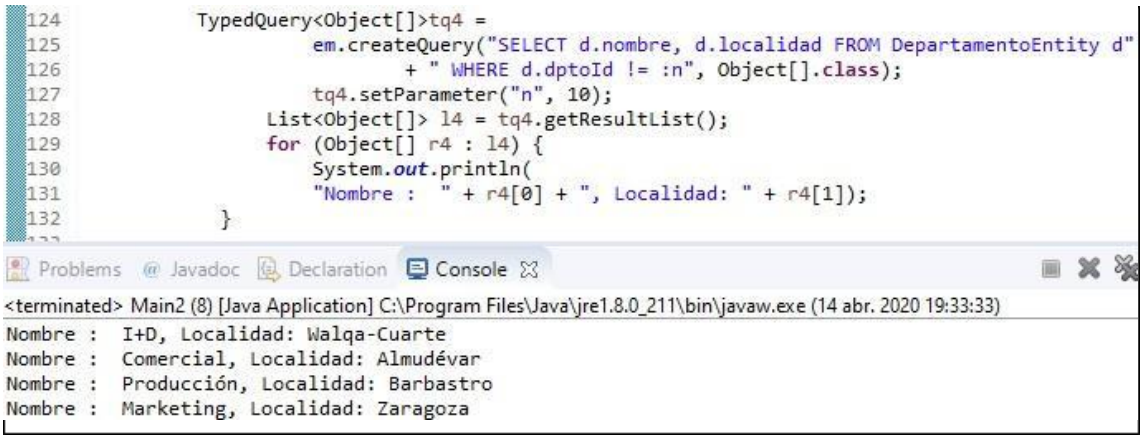


Ilustración 174

Ejercicio 8.- Realiza las siguientes consultas utilizando **el lenguaje JPQL**. Puedes incluirlas dentro del método *demoJPQL()* de la clase *AccesoBdatos*.

1. **Nombre y fecha de alta de todos los empleados**

```
Raquel Carrera Bailín - Tue Nov 12 00:00:00 CET 2002 -
Alberto Carrera Martín - Wed Oct 27 00:00:00 CEST 1999 -
Mario Carrera Bailín - Fri Jul 06 00:00:00 CEST 2001 -
Araceli Carrera Salcedo - Mon Feb 24 00:00:00 CET 2003 -
Blanca Bailín Perarnau - Thu Jul 12 00:00:00 CEST 2001 -
Fernando Carrera Martín - Mon Nov 19 00:00:00 CET 2001 -
Fernando Carrera Salcedo - Sun Feb 13 00:00:00 CET 2005 -
Carmen Bailín Perarnau - Fri Oct 19 00:00:00 CEST 2001 -
Javier Escartín Nasarre - Sun Jul 13 00:00:00 CEST 2003 -
Lorenzo Blasco González - Wed Mar 07 00:00:00 CET 2001 -
Enrique Casado Alvarez - Tue Jul 23 00:00:00 CEST 2002 -
Antonio Mériz Piedrafita - Sat Mar 22 00:00:00 CET 2003 -
Fernando Martínez Pérez - Thu Feb 03 00:00:00 CET 2000 -
Belén Carrera Sausán - Tue Feb 22 00:00:00 CET 2000 -
```

2. **Ídem de la anterior pero para aquellos que "Carrera" forma parte del nombre. No distinguir mayúsculas de minúsculas**

```
Raquel Carrera Bailín - Tue Nov 12 00:00:00 CET 2002 -
Alberto Carrera Martín - Wed Oct 27 00:00:00 CEST 1999 -
Mario Carrera Bailín - Fri Jul 06 00:00:00 CEST 2001 -
Araceli Carrera Salcedo - Mon Feb 24 00:00:00 CET 2003 -
Fernando Carrera Martín - Mon Nov 19 00:00:00 CET 2001 -
Fernando Carrera Salcedo - Sun Feb 13 00:00:00 CET 2005 -
Belén Carrera Sausán - Tue Feb 22 00:00:00 CET 2000 -
```

3. **Empleados del Departamento I+D cuyo oficio es el de Empleado**

```
Fernando Carrera Martín - Empleado - I+D -
Fernando Carrera Salcedo - Empleado - I+D -
```

4. **Empleados contratados a partir del 2003**

```
Araceli Carrera Salcedo - Mon Feb 24 00:00:00 CET 2003 -
Fernando Carrera Salcedo - Sun Feb 13 00:00:00 CET 2005 -
Javier Escartín Nasarre - Sun Jul 13 00:00:00 CEST 2003 -
Antonio Mériz Piedrafita - Sat Mar 22 00:00:00 CET 2003 -
```

5. **Empleados por orden alfabético de departamento**

```
Comercial - Javier Escartín Nasarre -
Comercial - Lorenzo Blasco González -
Comercial - Enrique Casado Alvarez -
Comercial - Antonio Mériz Piedrafita -
Comercial - Fernando Martínez Pérez -
Comercial - Belén Carrera Sausán -
Finanzas - Raquel Carrera Bailín -
Finanzas - Alberto Carrera Martín -
Finanzas - Mario Carrera Bailín -
I+D - Araceli Carrera Salcedo -
I+D - Blanca Bailín Perarnau -
I+D - Fernando Carrera Martín -
I+D - Fernando Carrera Salcedo -
I+D - Carmen Bailín Perarnau -
```

6. **Nombre, nº de empleados, total y máximo salario de los departamentos con empleados**

```
Comercial - 6 - 15635 - 3150 -
I+D - 5 - 14140 - 3000 -
Finanzas - 3 - 10975 - 4900 -
```

7. **Ídem de la anterior pero para departamentos a partir de 5 empleados**

```
Comercial - 6 - 15635 - 3150 -
I+D - 5 - 14140 - 3000 -
```

8. **Cada empleado junto con su jefe**

```
Raquel Carrera Bailín - su jefe es - Mario Carrera Bailín - departamento - 10 -
Mario Carrera Bailín - su jefe es - Alberto Carrera Martín - departamento - 10 -
Araceli Carrera Salcedo - su jefe es - Blanca Bailín Perarnau - departamento - 20 -
Blanca Bailín Perarnau - su jefe es - Alberto Carrera Martín - departamento - 20 -
Fernando Carrera Martín - su jefe es - Araceli Carrera Salcedo - departamento - 20 -
Fernando Carrera Salcedo - su jefe es - Carmen Bailín Perarnau - departamento - 20 -
Carmen Bailín Perarnau - su jefe es - Blanca Bailín Perarnau - departamento - 20 -
Javier Escartín Nasarre - su jefe es - Fernando Martínez Pérez - departamento - 30 -
Lorenzo Blasco González - su jefe es - Fernando Martínez Pérez - departamento - 30 -
Enrique Casado Alvarez - su jefe es - Fernando Martínez Pérez - departamento - 30 -
Antonio Mériz Piedrafita - su jefe es - Fernando Martínez Pérez - departamento - 30 -
Fernando Martínez Pérez - su jefe es - Alberto Carrera Martín - departamento - 30 -
Belén Carrera Sausán - su jefe es - Fernando Martínez Pérez - departamento - 30 -
```

9. **Nombre y total de empleados de los departamentos con algún empleado**

```
Comercial - 6 -
I+D - 5 -
Finanzas - 3 -
```

10. **Nombre y total de empleados de TODOS los departamentos**

```
Marketing - 0 -
Producción - 0 -
Comercial - 6 -
I+D - 5 -
Finanzas - 3 -
```

11. **Ordenando descendentemente por departamento y ascendentemente por salario**

```
30 - Lorenzo Blasco González - 2350 -
30 - Javier Escartín Nasarre - 2435 -
30 - Belén Carrera Sausán - 2500 -
30 - Enrique Casado Alvarez - 2600 -
30 - Antonio Mériz Piedrafita - 2600 -
30 - Fernando Martínez Pérez - 3150 -
20 - Carmen Bailín Perarnau - 2600 -
20 - Fernando Carrera Salcedo - 2730 -
20 - Fernando Carrera Martín - 2840 -
20 - Blanca Bailín Perarnau - 2970 -
20 - Araceli Carrera Salcedo - 3000 -
10 - Raquel Carrera Bailín - 2690 -
10 - Mario Carrera Bailín - 3385 -
10 - Alberto Carrera Martín - 4900 -
```

12. **Empleados sin jefe.**

```
1039 - Alberto Carrera Martín -
```

13. **Departamento al que pertenece el empleado nº 1039**

```
10 - Finanzas -
```

Ejercicio 9.- Realiza los siguientes métodos (dentro de de la clase *AccesoBdatos*) utilizando **el lenguaje JPQL**.

1. Método *public int incrementarSalario (int cantidad)* para incrementar el salario de todos los empleados en la cantidad pasada como argumento. Utiliza la sentencia UPDATE con parámetros. El método devuelve el número de filas modificadas.

2. Método *public int incrementarSalarioOficio (String oficio, int cantidad)*. Ídem del anterior pero solo para los de un determinado oficio.

3. Método *public int incrementarSalarioDepartamento (int numDepartamento, int cantidad)*. Ídem del anterior pero solo para los empleados de un departamento concreto.

4. Método *public int borrarEmpleado (int numEmpleado)* para borrar el empleado que se pasa como argumento. (Nota: Después de hacerlo comprueba el método anterior de tal forma que si el empleado borrado es jefe de algún otro empleado, este último pasará a contener NULL en su atributo dirId). Utiliza la sentencia DELETE con parámetros. El método devuelve el número de filas borradas.

5. Método *public int borrarDepartamento(int numDepartamento)* para borrar el departamento que se pasa como argumento. (Nota: Si el departamento borrado tiene empleados a su cargo, debido a la relación establecida en el momento de creación de la entidad, no se borra ninguno de éstos, dejando además el atributo departamento de sus empleados con todos sus valores puestos a NULL excepto el dptoId que conserva su valor aunque corresponda a un departamento que ya no existe). Utiliza la sentencia DELETE con parámetros. El método devuelve el número de filas borradas.

SESIÓN 4

Recupera el proyecto *XObjectDbSesion4* que se encuentra en la carpeta *ficheros*. Verás que se trata de un proyecto muy parecido a otro anterior con algunas modificaciones que pasaremos a tratar.

Ejercicio 10.- Investiga y averigua las cuestionas que se plantean a continuación. Puedes anotar las respuestas en un fichero Word. En la anotación de la relación de la ilustración siguiente (líneas 19 a 20):

a. ¿Qué significa el parámetro *cascade*?
Es la manera de indicar que vamos a incluir un parámetro de actualización en cascada a la clase asociada en la relación, en este caso, en la relación. **OneToMany** (Cada vez que se haga algo en la clase **DepartamentoEntity**, se propagara a la clase asociada **EmpleadoEntity**)

b. ¿Qué significa y qué otras posibilidades hay además de la opción *CascadeType.ALL*?
Con **CascadeType.ALL** indicamos que cualquiera de estas operaciones **(PERSIST, REMOVE, REFRESH, MERGE, DETACH)** en la clase **DepartamentoEntity**, se propagaran a la clase asociada (**EmpleadoEntity**) en la relación **OneToMany** que tiene con **EmpleadoEntity**.

Las otras opciones disponibles son: CascadeType.(ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH)

c. ¿Qué significa la opción *orphanRemoval=true*?
Indica que en el caso de eliminar un **DepartamentoEntity**, si tiene empleados asociados **(EmpeladoEntity)**, se eliminarán también dichos empleados. De manera transparente a nosotros sin tener que preocuparnos, y si lo pusieramos en false. Cuando intentamos eliminar un Departamento que contiene empleados, saltará un error, porque detectaría la relación existente entre estas dos clases y que hay empleados en ese departamento.

d. ¿Qué significa y qué otras posibilidades hay además de la opción *fetch = FetchType.EAGER*?
Hay dos tipos de fetch (carga), **EAGER y LAZY (Ansiosa y Perezosa)**

Con la carga ansiosa(EAGER) lo que conseguimos es que cada vez que carguemos un DepartamentoEntity, al detectar que tiene una relación con EmpleadoEntity, se cargará el departamento con todos sus empleados, guardando toda esta información en memoria.

Al contrario que con la carga perezosa(LAZY), que lo que hace es que cuando carguemos una instancia de un departamento, cargará en memoria los datos del departamento pero no los de su relación con empleados, de esta manera ocupará menos en memoria. Para acceder a los datos de los empleados, habría que preguntarle por esos datos explícitamente en el momento que queramos. Pero nos ahorramos en memoria que no los haya cargado desde un principio.

*DepartamentoEntity.java

```
1  import java.util.HashSet;
9  //
10 //Alberto Carrera Martín - Abril 2020
11 // Revisado Febrero y Diciembre 2021
12 //
13 @Entity
14 public class DepartamentoEntity {
15     @Id
16     private int dptoId;
17     private String nombre;
18     private String localidad;
19     @OneToMany(mappedBy="departamento", cascade= CascadeType.ALL,
20             orphanRemoval=true, fetch = FetchType.EAGER)
21         private Set<EmpleadoEntity> empleados = new HashSet<EmpleadoEntity>();
22     //
23     public DepartamentoEntity(int dptoId, String nombre, String localidad) {
24
25         this.dptoId = dptoId;
```

Ejercicio 11.- Investiga y averigua las cuestiones que se plantean a continuación referidas a la siguiente ilustración. Puedes anotar las respuestas en un fichero Word.

EmpleadoEntity.java

```
12  //
13  //Alberto Carrera Martín - Abril 2020
14  // Revisado Febrero 2021
15
16  @NamedQueries({
17          @NamedQuery(name = "numeroEmpleados",
18              query = "SELECT count(e) FROM EmpleadoEntity e "
19                    + "WHERE e.departamento.nombre = :nombre"),
20          //@NamedQuery(name=".....
21      })
22
23  @Entity
24  public class EmpleadoEntity {
25      @Id
26      private int empnoId;
27      private String nombre;
28      private String oficio;
29      private EmpleadoEntity dirId;
30      private Date alta;
31      private Integer salario;
32      private Integer comision;
33      @ManyToOne
34      @JoinColumn(name="dptoId")
35          private DepartamentoEntity departamento;
36      //
        ••••••••
```

a. ¿Qué significa la anotación *@JoinColumn* de la línea 34? ¿Es obligatoria?

Esta anotación sirve en JPA para hacer referencia a la columna que es **clave externa en la tabla** y que se encarga de definir la relación. Sirve para personalizar la relación, le indicamos explícitamente que esa columna (departamento) será utilizada para la unión.

En realidad cuando indicamos las anotaciones MenyToOne, … JPA se da cuenta de las relaciones y de manera transparente a nosotros recrea la union de tablas con las claves foráneas(join). Aunque si no queremos estar a la merced de JPA, deberíamos indicar nosotros de manera manual, que columna es la que se tiene en cuenta para la reacción JOIN. Es por eso que indicamos JoinColumn, aun que como digo, en principio esto JPA lo hace de manera automática por nosotros aunque no indiquemos esta anotación.

b. ¿Qué se ha creado en las líneas 16 a 19 anteriores? ¿Con qué finalidad? (pista: clase *MainNamedQueries.java* del proyecto.

Como en un proyecto de verdad, es posible que diferentes programadores creen sus propias clases (Acceso a datos) donde crear consultas que ataquen a las clases (DepartamentoEntity, …), lo que pasará es que acabaremos con un montón de consultas duplicadas. Cada programador habrá creado sus consultas, y seguro que algunas de ellas coincidan.

Para evitar eso, podemos crear consultas ya preparadas en las clases(tablas) de la base de datos.

Es decir en la clase (Tabla) EmpleadoEntity podemos definir una consulta con un id y el programador solo tendrá que llamar a esa consulta con su id y se ejecutará. Con esto nos ahorramos que se creen muchas consultas duplicadas y asi las consultas comunes ya esten grabadas en la tabla. Son como consultas preparadas. **En este caso la consulta ya preparada le hemos puesto el nombre de numeroEmpleados.**

Ejercicio 12.- Realiza el mismo proyecto que el del ejercicio 1 del boletín *04_Mapeo_objeto_relacional_Ejercicios.pdf* pero esta vez adaptándolo a la base de datos orientada a objetos ObjectDB.

Ejercicio 13.- Realiza el mismo proyecto que el del ejercicio 3 del boletín *04_Mapeo_objeto_relacional_Ejercicios.pdf* pero esta vez adaptándolo a la base de datos orientada a objetos ObjectDB.

SESIÓN 5

Ejercicio 14.-

1. Busca y adapta a ObjectDB un proyecto JPA que trabaje con dos entidades en las que exista una relación uno a uno (unidireccional o bidireccional).
2. Idem de antes pero una relación muchos a muchos (unidireccional o bidireccional).
3. Idem de antes pero con una relación de herencia entre las entidades (estrategia de tabla única).
   Las relaciones de herencia en JPA se pueden aplicar de 3 maneras: **SINGLE_TABLE**, **JOINED_TABLE** y **TABLE_PER_CONCRETE_CLASS**. En este caso, el ejercicio nos pide resolverla con la forma de **SINGLE_TABLE.**

   https://www.tutorialspoint.com/es/jpa/jpa_advanced_mappings.htm#:~:text=La%20herencia%20es%20el%20concepto,%3A%20SINGLE_TABLE%2C%20JOINED_TABLE%20y%20TABLE_PER_CONCRETE_CLASS.

   Para entender la anotación de discriminacion:
   https://www.arquitecturajava.com/jpa-single-table-inheritance/