

INDICE

1.1 BASES DE DATOS ORIENTADA A OBJETOS	1
1.2 OBJECTDB	2
1.2.1 ESTRUCTURA DE OBJECTDB	2
1.3 LA API DE PERSISTENCIA DE JAVA: JPA	3
1.3.1 ¿QUÉ ES JPA?	3
1.3.2 ¿CÓMO NACE JPA?	4
1.3.3 ¿EL OBJETIVO DE JPA?	5
1.3.4 ¿DÓNDE ENCAJA JPA?	5
1.3.5 ¿JPA E HIBERNATE?	5
1.3.6 ¿CÓMO FUNCIONA JPA?	6
1.3.6.1 MAPEADO DE ENTIDADES	7
1.3.6.2 TIPOS DE DATOS PERSONALIZADOS	9
1.3.6.3 JPQL Y CONSULTAS NATIVAS	9
1.3.6.4 OTRAS CARACTERÍSTICAS	9
1.3.7 IMPLEMENTACIONES DE JPA	10
1.3.8 COMENZAR A UTILIZAR JPA CON NETBEANS	10
1.3.9 EXPLORAR LA BASE DE DATOS USANDO EL EXPLORADOR DE OBJECTDB	11
1.4 JPA VS JDBC	14
1.5 REFERENCIAS:	15

1.1 Bases de Datos Orientada a Objetos

Las Bases de Datos Orientadas a Objetos (BDOO) son aquellas cuyo modelo de datos está orientado a objetos, soportan el paradigma orientado a objetos almacenando métodos y datos. Su origen se debe principalmente a la existencia de problemas para representar cierta información y modelar ciertos aspectos del mundo real. Las BDOO simplifican la programación orientada a objetos (POO) y almacenando directamente los objetos en la BD y empleando las mismas estructuras y relaciones que los lenguajes de POO.

Podemos decir que un Sistema Gestor de Base de Datos Orientada a Objetos (SGBDOO) es un sistema gestor de base de datos (SGBD) que almacena objetos.

1.2 ObjectDB

En los contenidos de la unidad se mencionan otros gestores de bases de datos como db4o, Matisse....éstos están prácticamente en desuso y con poca documentación disponible, entonces para desarrollar las tareas prácticas trabajaremos con el gestor de bases de datos orientado a objetos, ObjectDB.

ObjectDB es un potente sistema de gestión de bases de datos orientado a objetos (ODBMS). Es compacto, confiable, fácil de usar y extremadamente rápido. ObjectDB proporciona todos los servicios de administración de bases de datos estándar (almacenamiento y recuperación, transacciones, administración de bloqueos, procesamiento de consultas, etc.) pero de una manera que facilita el desarrollo y agiliza las aplicaciones.

Se puede utilizar en modo cliente-servidor y en modo incrustado (en proceso).

A diferencia de otras bases de datos orientadas a objetos, ObjectDB no proporciona su propia API propietaria. Por lo tanto, el trabajo con ObjectDB requiere el uso de una de las dos API estándar de Java - JPA o JDO. Ambas APIs están incorporadas en ObjectDB. Nosotros usaremos el estándar JPA, que veremos más adelante.

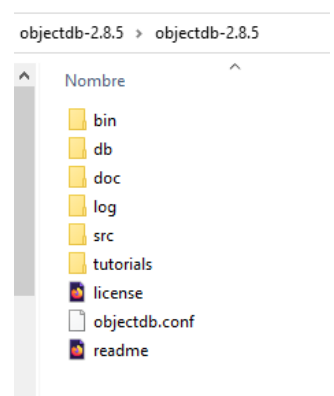
Referencia: <https://www.objectdb.com/>

1.2.1 Estructura de ObjectDB

Se trata de un zip que se puede [descargar](#) de la página web del proveedor, es una licencia open source y todos los detalles de la licencia se pueden encontrar en la documentación.

Con respecto al sistema de carpetas podemos encontrar las siguientes:

- **bin**, se puede decir que es prácticamente el cuerpo de la base de datos y contiene diversas herramientas. Dispone de un explorador para visualizar y hacer queries en la base datos, algo similar a lo que ofrece phpmyadmin con MySQL.
- **db**, carpeta donde se almacenan las bases de datos.
- **doc**, carpeta donde todas las funcionalidades de la versión que se está utilizando, en mi caso la versión 2.8.
- **log**, carpeta donde se almacenan los registros de todas las operaciones que se realizan en el sistema.
- **src**, carpeta donde se ubican las librerías para la manipulación de datos usando *jdo* y *jpa*.
- **tutorials**, carpeta que contiene varios tutoriales incluso la posibilidad de crear una página web usando base de datos orientada a objetos.
- el **readme**, la información de la licencia y la configuración el archivo de configuración.



1.3 La API de persistencia de Java: JPA

Casi cualquier aplicación que vayamos a construir, tarde o temprano tiene que lidiar con la **persistencia de sus datos**. Es decir, debemos lograr que los datos que maneja o genera la aplicación **se almacenen fuera de esta para su uso posterior**.

Esto, por regla general, implica el uso de un **sistema de base de datos**, bien tradicional basado en SQL, bien de tipo documental (también conocido como No-SQL).

A la hora de desarrollar, los programas modernos modelan su **información utilizando objetos** (Programación Orientada a Objetos), pero **las bases de datos utilizan otros paradigmas**: las relaciones (también llamadas a veces "Tablas") y las claves externas que las relacionan. Esta diferencia entre la forma de programar y la forma de almacenar da lugar a lo que se llama **"desfase de impedancia"** y complica la persistencia de los objetos.

Para minimizar ese desfase y facilitar a los programadores la persistencia de sus objetos de manera transparente, nacen los denominados ORMs o mapeadores Objeto-Relacionales que ya vimos en la unidad anterior.

1.3.1 ¿Qué es JPA?

Ahora que ya estamos más o menos situados, podemos empezar a ver **qué es la API de Persistencia de Java o JPA**.

Nota: aunque API de persistencia era su nombre original y, en realidad, el que todo el mundo sigue usando, conviene saber que el nombre oficial ahora es *Jakarta Persistence*, ya que Jakarta es el nombre actual de lo que antiguamente se llamaba *Java Platform Enterprise Edition* o Java EE (también llamada coloquialmente "Java empresarial"). El motivo es que en 2018 pasó a gestionarlo la fundación Eclipse y, como Java es una marca registrada de Oracle, se vieron forzados también a cambiarle el nombre, decidiéndose por "Jakarta".

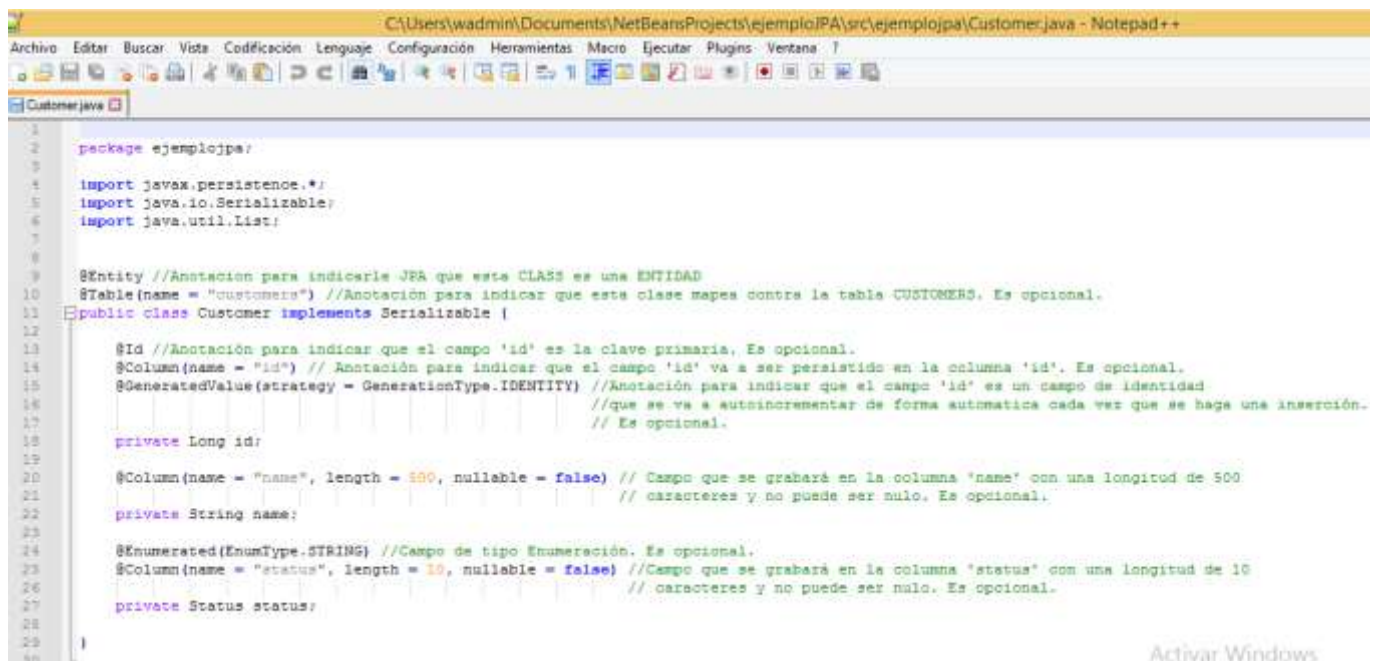
JPA es **una especificación** que indica cómo se debe realizar la persistencia (almacenamiento) de los objetos en programas Java. Fíjate en que destaco la palabra "Especificación" porque JPA no tiene una implementación concreta, sino que, como veremos enseguida, existen diversas tecnologías que implementan JPA para darle concreción.

- ✓ JPA fue lanzado por primera vez en **2006**, comparte de **Enterprise JavaBeans (EJB) 3.0**.
- ✓ Fue sin duda, una de las tecnologías más emocionantes de **Java EE 5**. Como sabemos se puede persistir con otras tecnologías como JDBC que aun teniendo sus ventajas, como su rapidez, presenta ciertas limitaciones, es más complicado y enredoso de usar y sobretodo inyecta muchos errores en tiempo de ejecución.
- ✓ JPA surge aprovechando las ventajas que ofrecía el JDK6 que eran las **anotaciones**. Antes de las anotaciones lo que se tenía eran archivos XML donde se describían como se tenían que mapear los objetos y clases.

Desde entonces han aparecido varias versiones de la especificación, que sigue el proceso de la comunidad Java, siendo la más reciente la 2.2, aparecida en el verano de 2017.

Aunque forma parte de Java empresarial, las implementaciones de JPA **se pueden emplear** en cualquier tipo de aplicación aislada, sin necesidad de usar ningún servidor de aplicaciones, **como una mera biblioteca de acceso a datos**.

Ejemplo:



```
1 package ejemplojpa;
2
3
4 import javax.persistence.*;
5 import java.io.Serializable;
6 import java.util.List;
7
8
9 @Entity //Anotación para indicarle JPA que esta CLASS es una ENTIDAD
10 @Table(name = "customers") //Anotación para indicar que esta clase mapea contra la tabla CUSTOMERS. Es opcional.
11 public class Customer implements Serializable {
12
13     @Id //Anotación para indicar que el campo 'id' es la clave primaria. Es opcional.
14     @Column(name = "id") // Anotación para indicar que el campo 'id' va a ser persistido en la columna 'id'. Es opcional.
15     @GeneratedValue(strategy = GenerationType.IDENTITY) //Anotación para indicar que el campo 'id' es un campo de identidad
16     //que se va a autoincrementar de forma automática cada vez que se haga una inserción.
17     // Es opcional.
18     private Long id;
19
20     @Column(name = "name", length = 500, nullable = false) // Campo que se grabará en la columna 'name' con una longitud de 500
21     // caracteres y no puede ser nulo. Es opcional.
22     private String name;
23
24     @Enumerated(EnumType.STRING) //Campo de tipo Enumeración. Es opcional.
25     @Column(name = "status", length = 10, nullable = false) //Campo que se grabará en la columna 'status' con una longitud de 10
26     // caracteres y no puede ser nulo. Es opcional.
27     private Status status;
28
29
30 }
```

1.3.2 ¿Cómo nace JPA?

JPA no surge de la nada, hay varias herramientas que preceden a JPA e inspiraron su desarrollo y son las siguientes:

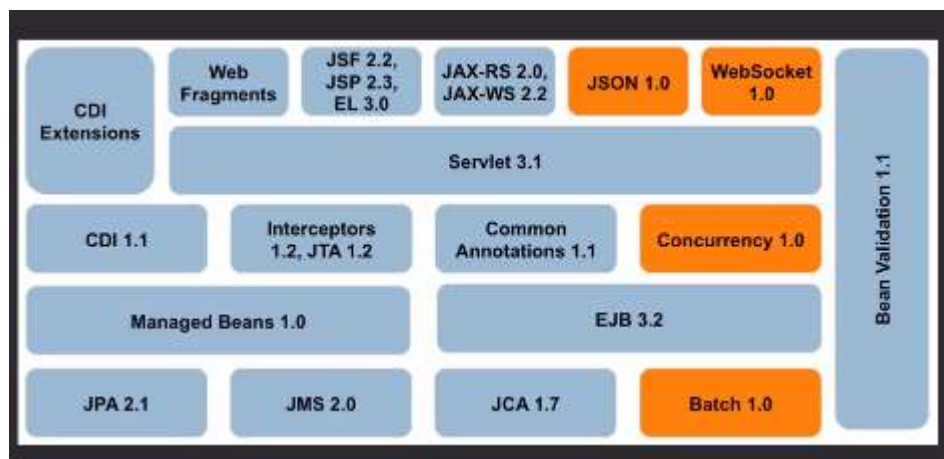
- ✓ **Java JDBC**, es la base sobre la que está montada JPA, en la actualidad se sigue utilizando, es la forma más nativa de trabajar con la base de datos y la que proporciona las herramientas de más bajo nivel para trabajar con la base de datos como por ejemplo lanzar queries nativas (select, insert, update...).
- ✓ **Entity Beans**, en la actualidad están en desuso pero se utilizaron con Java durante muchos años. Los Entity Beans formaban parte de los EJB 2.0, eran entidades o beans que podían ser persistentes pero eran muy complicadas de utilizar. No entraremos en más detalles ya que desaparecieron y no tiene caso dedicarle más tiempo.
- ✓ **Hibernate**, fue el pionero de los ORM, es la herramienta más potente para realizar la persistencia. Inicialmente la persistencia la hacía usando archivos XML aunque era complicado era la única alternativa a los Entity Beans y EJB2.0.
- ✓ **Ibatis**, era la competencia de Hibernate como ORM's principales.

1.3.3 ¿El objetivo de JPA?

- ✓ Reemplazar a los **Entity Beans**, la cual era una tecnología muy compleja.
- ✓ Crear una **especificación** compatible con EJB 3.0 capaz de ser ejecutada fuera de aplicaciones empresariales.
- ✓ Crear un **API Estándar** para la persistencia de objetos para evitar que los desarrollares tuvieran que conocer diferentes frameworks dependiendo del ORM que utilizarasen para trabajar con persistencia en Java, en ese momento cada ORM iba por libre. Resumiendo, JPA viene a estandarizar el uso de los ORM's.

1.3.4 ¿Dónde encaja JPA?

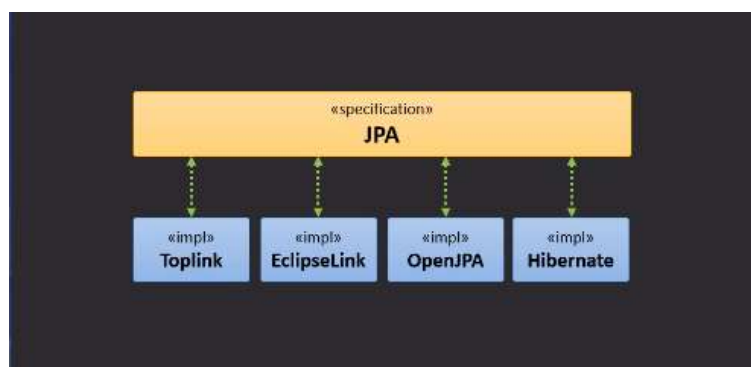
Java Enterprise Edition se conforma de una serie de especificaciones. Una **especificación** es un documento donde se redacta como las tecnologías deben de funcionar sin ser código. Es decir, los desarrolladores de las especificaciones proponen como ciertas herramientas deben de trabajar por ejemplo, qué interfaces debe tener, qué clases debe incluir.....



Como se puede ver en la imagen para todas las tecnologías de Java existe una especificación.

1.3.5 ¿JPA e Hibernate?

Es muy común no tener clara la diferencia entre JPA e Hibernate. Bien, teniendo claro que JPA es una especificación e Hibernate dispone de un framework que trabaja de forma NO estándar, Hibernate lo que hace es implementar las directrices que dicta la especificación JPA para que trabaje de forma compatible.



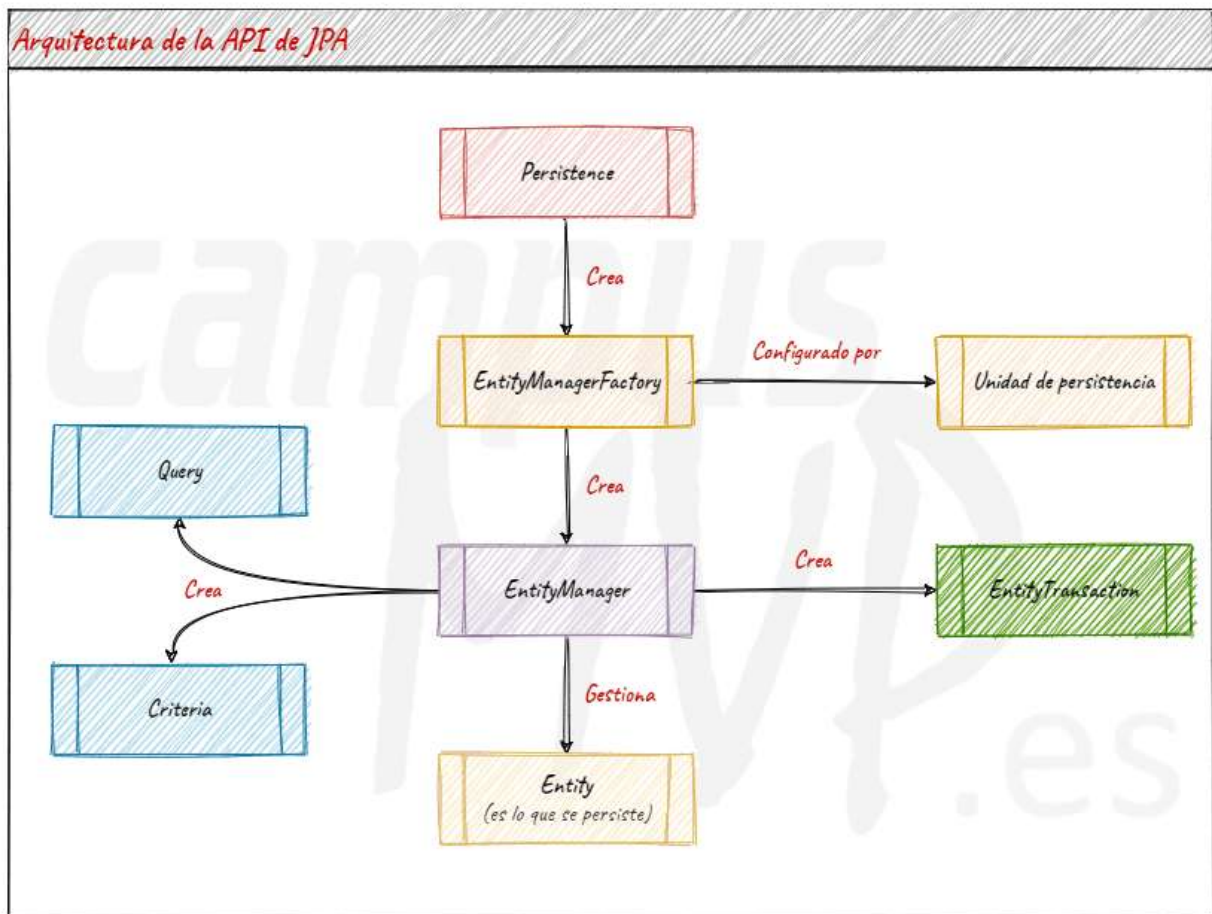
- ✓ TopLink, implementación de Oracle.
- ✓ EclipseLink, implementación de la Fundación Eclipse.
- ✓ OpenJPA, implementación de Apache.
- ✓ Hibernate, implementación de RedHat de IBM.
- ✓ ObjectDB, implementación desarrollada por ObjectDB Software.

Conclusiones

- ✓ JPA es el API **estándar** que ofrece Java para la persistencia de objetos (ORM).
- ✓ JPA nace de la evolución de los **Entity Beans**, que solo funcionaban en entornos empresariales.
- ✓ JPA es una **especificación**, por lo que existen múltiples implementaciones, entre ellas Hibernate.
- ✓ JPA utiliza el API **JDBC** por debajo.
- ✓ JPA fue inspirada en varios ORM's pero sobre todo en **Hibernate**.

1.3.6 ¿Cómo funciona JPA?

Dado que es una especificación, **JPA no proporciona clase alguna para poder trabajar** con la información. Lo que hace es proveernos de una serie de interfaces que podemos utilizar para implementar la capa de persistencia de nuestra aplicación, apoyándonos en alguna implementación concreta de JPA.



Es decir, **en la práctica significa que lo que vas a utilizar es una biblioteca de persistencia que implemente JPA**, no JPA directamente. Más adelante, veremos algunas de estas implementaciones, pero, mientras tanto, conviene saber qué funcionalidades definen estas interfaces en la especificación, mediante las interfaces del diagrama anterior, ya que luego serán las que utilices con cada implementación concreta.

1.3.6.1 Mapeado de entidades

Lo primero que hay que hacer para usar una implementación de JPA es añadirla al proyecto y configurar el sistema de persistencia (por ejemplo, la cadena de conexión a una base de datos). Pero, tras esas cuestiones de "fontanería", lo más básico que deberás hacer y que es el núcleo de la especificación es el **mapeado de entidades**.

El "mapeado" se refiere a definir cómo se relacionan las clases de la aplicación con los elementos del nuestro sistema de almacenamiento.

Si consideramos **el caso común de acceso a una base de datos relacional**, sería definir:

- La relación existente **entre las clases** de tu aplicación **y las tablas** de tu base de datos

- La relación **entre las propiedades** de las clases **y los campos** de las tablas
- La relación **entre diferentes clases y las claves externas** de las tablas de la base de datos

Esto último es muy importante, ya que permite definir propiedades en nuestras clases que son otras clases de la aplicación, y obtenerlas automáticamente desde la persistencia en función de su relación. Mejor veámoslo con un ejemplo:

```
1  @Entity
2  public class Factura {
3      @ManyToOne
4      private List<LineaFactura> lineasFactura;
5      ...
6  }
```

En este ejemplo, cada factura consta de varias líneas de factura, y con estos atributos estamos definiendo esta circunstancia. Posteriormente, cuando obtengamos una factura podremos consultar sus líneas de factura de manera transparente, pues será la implementación concreta de JPA la que se encargue de obtener automáticamente las asociadas a esa factura desde la base de datos sabiendo de esta relación.

Como ves, esto facilita enormemente el acceso a datos.

Nota: aunque esto es muy cómodo, es necesario tener mucho cuidado con los tipos de relaciones que definimos y de qué manera y dónde lo hacemos, ya que puede influir negativamente en el rendimiento de nuestra aplicación.

Como puedes observar en el fragmento anterior, lo que define JPA es también una serie de **anotaciones** con las que podemos decorar nuestras clases, propiedades y métodos para indicar esos "mapeados".

Además, JPA simplifica aún más el trabajo gracias al uso de una serie de **convenciones por defecto** que sirven para un alto número de casos de uso habituales. Por ello, **solo deberemos anotar aquellos comportamientos que queramos modificar** o aquellos que no se pueden deducir de las propias clases. Por ejemplo, aunque existe una anotación (**@Table**) para indicar el nombre de la tabla en la base de datos que está asociada a una clase, por defecto si no indicamos otra cosa se considerará que ambos nombres coinciden. Por ejemplo, en el fragmento anterior, al haber anotado con **@Entity** la clase Factura se considera automáticamente que la tabla en la base de datos donde se guardan los datos de las facturas se llama también *Factura*. Pero podríamos cambiarla poniéndole la anotación **@Table(name="Invoices")**, por ejemplo.

Nota: también define una manera de hacerlo al revés, es decir, partir de una base de datos y generar a partir de esta las entidades de la aplicación, pero no se recomienda su uso.

1.3.6.2 Tipos de datos personalizados

Todos los tipos de datos fundamentales estándar están soportados por JPA. Pero, también es posible dar soporte a tipos de datos propios, personalizados, utilizando lo que se llaman convertidores, a través de la interfaz `AttributeConverter` y la anotación `@Converter`.

1.3.6.3 JPQL y consultas nativas

JPA define también su propio lenguaje de consultas llamado JPQL. Es similar a SQL y se utiliza para lanzar consultas específicas pero basadas en las entidades que tenemos en la aplicación y no en las tablas de la base de datos. Es decir, **las consultas JPQL hacen referencia a las clases Java y sus campos** en lugar de a tablas y columnas. Y además son **independientes del motor de bases de datos**.

A partir de estas consultas obtenemos ya directamente entidades de nuestra aplicación. Por ejemplo:

```
1 TypedQuery<Factura> consulta = modelo
2   .createQuery("SELECT f FROM Factura f WHERE f.id = :id", Factura.class);
3 consulta.setParameter("id", 5);
4 Factura f = consulta.getSingleResult();
```

También podemos lanzar consultas nativas a la base de datos:

```
1 Query consulta = modelo
2   .createNativeQuery("SELECT * FROM Factura WHERE id = :id", Factura.class);
3 consulta.setParameter("id", 5);
4 Factura f = (Factura) consulta.getSingleResult();
```

Las consultas JPQL son más limitadas que las que ofrece el lenguaje SQL, especialmente si consideramos las particularidades de los "dialectos" de cada motor de datos, pero son suficientes para un porcentaje alto de los casos. En cualquier caso, no deberíamos abusar de ninguno de los dos tipos de consultas, pues van en contra de la filosofía de los ORMs.

1.3.6.4 Otras características

Aparte de todo lo mencionado, a lo largo de los años y las versiones JPA han ido añadiendo soporte para más funcionalidades como:

- Soporte de colecciones y listas ordenadas.
- API Criteria para construcción de consultas.
- API adicional para generación de consultas DDL (creación y modificación de estructura de la base de datos).
- Soporte para validaciones.
- Ejecución de procedimientos almacenados.
- Etc...

Para profundizar en el tema podéis visitar la siguiente referencia.

- Referencias: [Especificación de JPA](#)

1.3.7 Implementaciones de JPA

Como se ha dicho anteriormente, JPA es una especificación, así que **para usarlo en la práctica se necesita una implementación concreta**, que implemente todas las interfaces y cuestiones definidas por la especificación.

La principal ventaja de esto es que, si las bibliotecas de persistencia que utilicemos siguen la especificación JPA, podremos **cambiar de una a otra**, con más rendimiento o características mejores, **sin tener que tocar el código**, simplemente cambiando las referencias.

Existen **diversas implementaciones** disponibles, como DataNucleus, ObjectDB, o Apache OpenJPA, pero las dos más utilizadas son **EclipseLink** y sobre todo **Hibernate**.

La implementación de referencia de JPA, es decir, la implementación que proporciona la propia Fundación Eclipse y que sirve para ver cómo crear un ORM basado en la especificación, **es EclipseLink**. Esta implementación incluye soporte para persistencia en bases de datos relacionales, bases de datos documentales (como MongoDB), almacenes XML (especificación JAXB) y servicios web de bases de datos, por lo que es muy completa. Además añade otras características propias por encima de JPA, como son eventos para reaccionar a cambios en bases de datos o poder mapear entidades en diferentes bases de datos mismo tiempo.

La otra gran implementación de JPA es Hibernate, que en la actualidad es casi el "estándar" *de facto*, puesto que **es la más utilizada**, sobre todo en las empresas. Es tan popular que existen hasta versiones para otras plataformas, como NHibernate para la plataforma .NET. Es un proyecto muy maduro (de hecho, la especificación JPA original partió de él), muy bien documentado y que tiene un gran rendimiento.

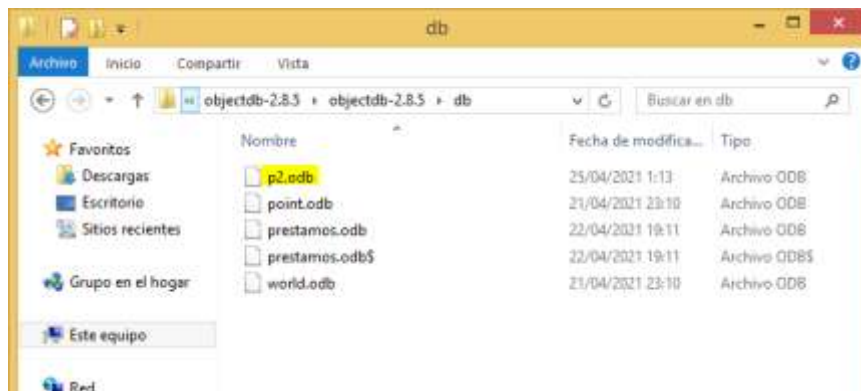
1.3.8 Comenzar a utilizar JPA con NetBeans

Para comenzar a utilizar JPA podéis probar a implementar el [ejemplo](#) que se muestra en el tutorial de la página oficial de ObjectDB. Donde se especifican todos los 4 pasos a seguir para crear el ejemplo.

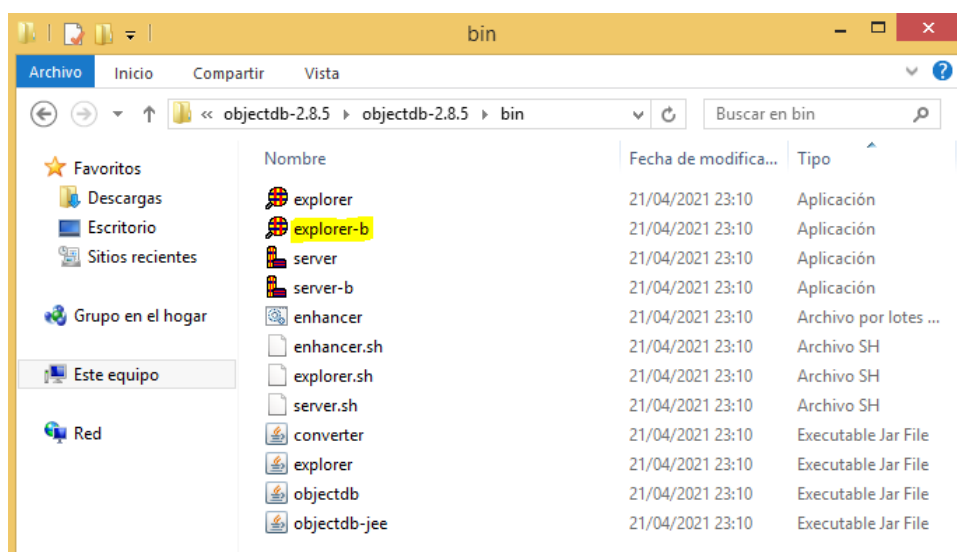


1.3.9 Explorar la base de datos usando el Explorador de ObjectDB

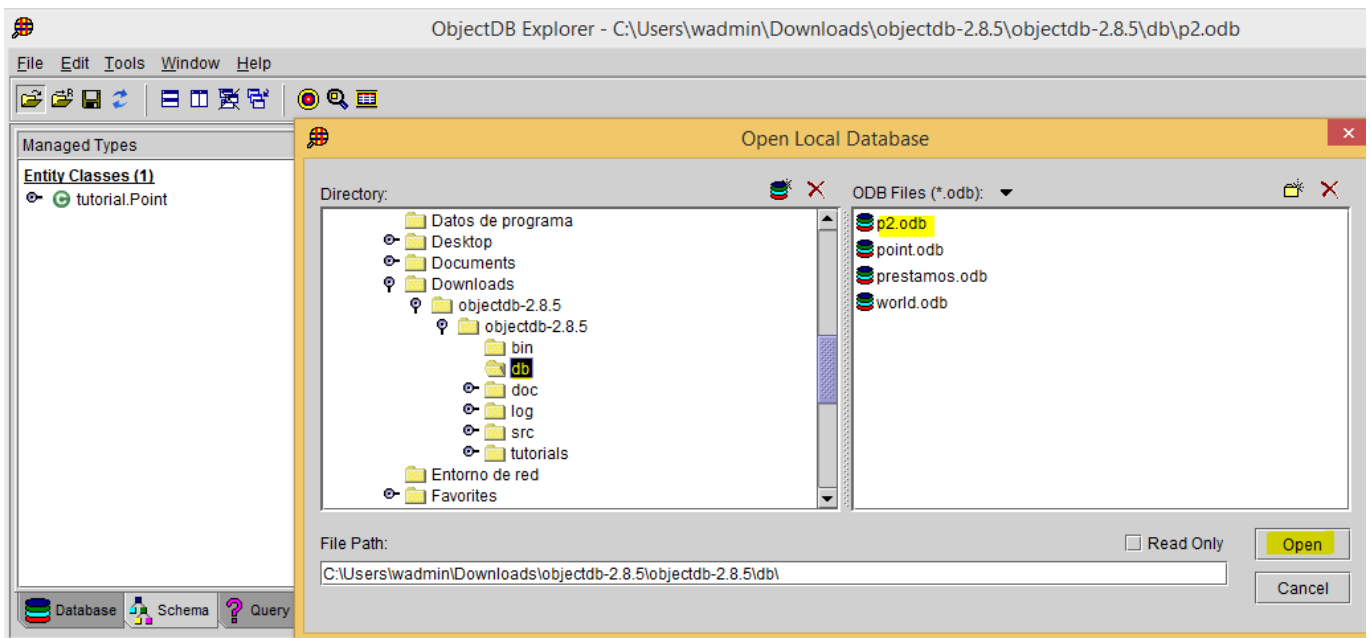
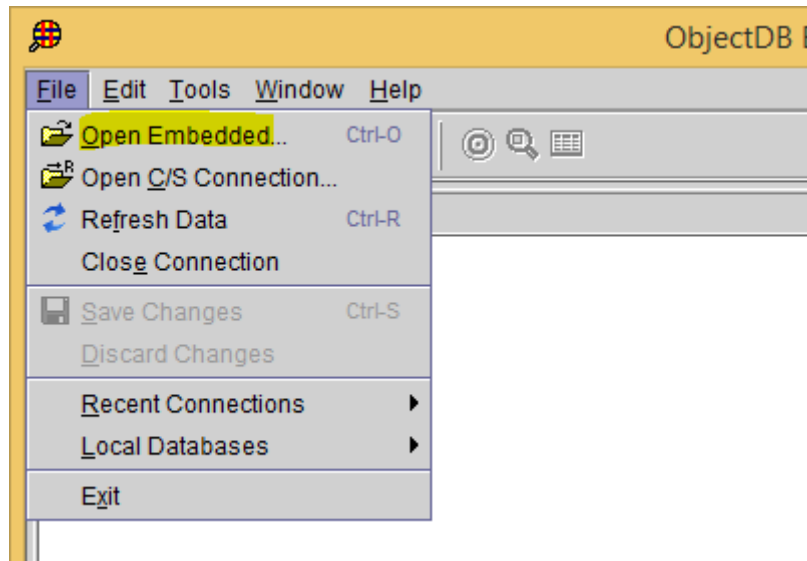
Una vez ejecutado el programa, se ha creado la BD **p2.oddb** que se encuentra en la carpeta '**db**'.




Para lanzar el Explorador, ejecutamos la aplicación '**explorer-b**' ubicado en la carpeta '**bin**'.

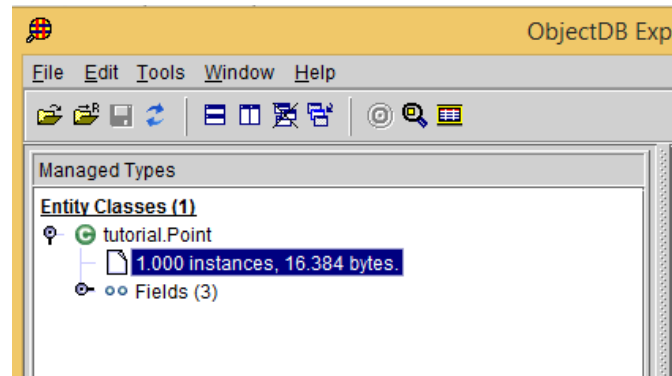


Buscamos la base de datos:

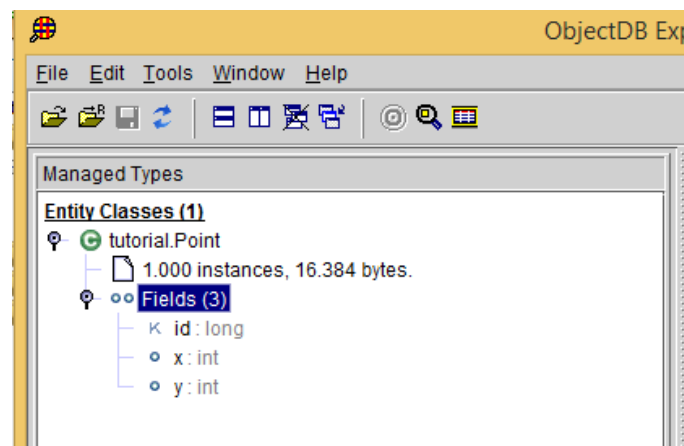


Si nos movemos por el árbol (haciendo clic en la llave ) podemos ver:

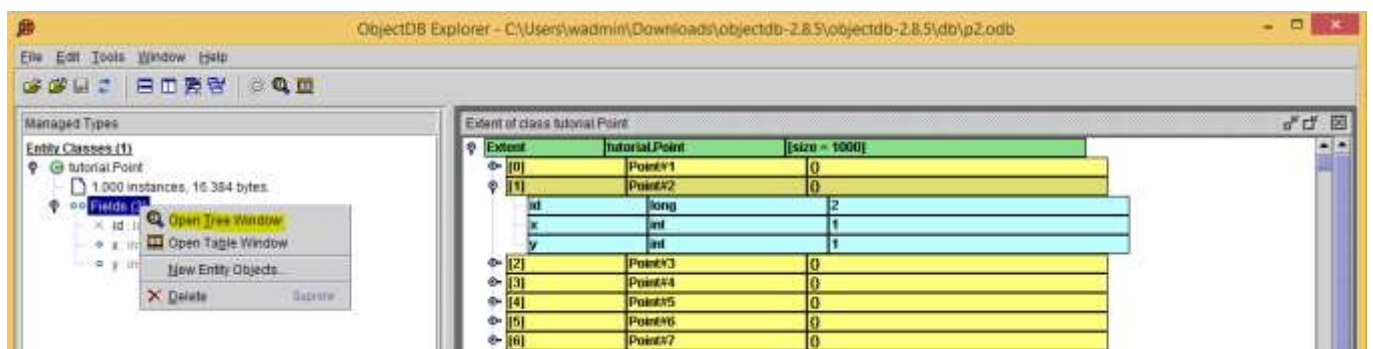
- ❖ El número de instancias almacenadas en la BD, en nuestro ejemplo 1000 instancias.



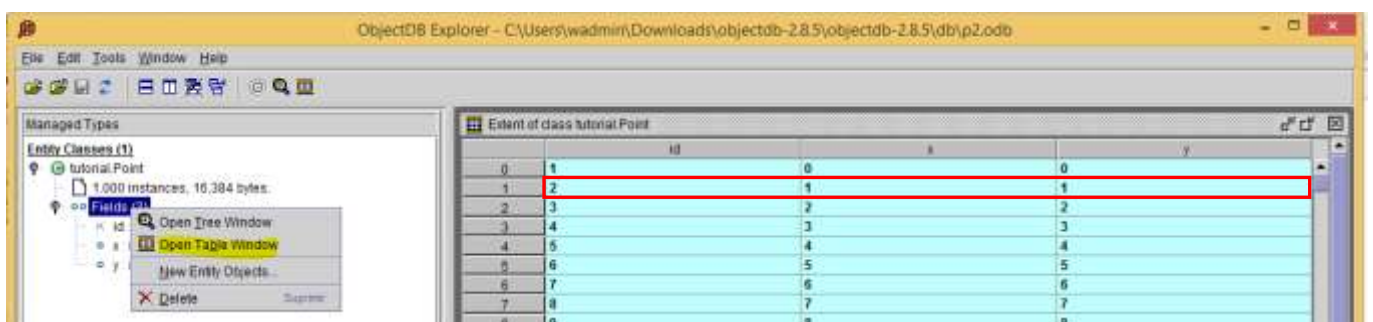
- ❖ Número y Tipo de los Campos



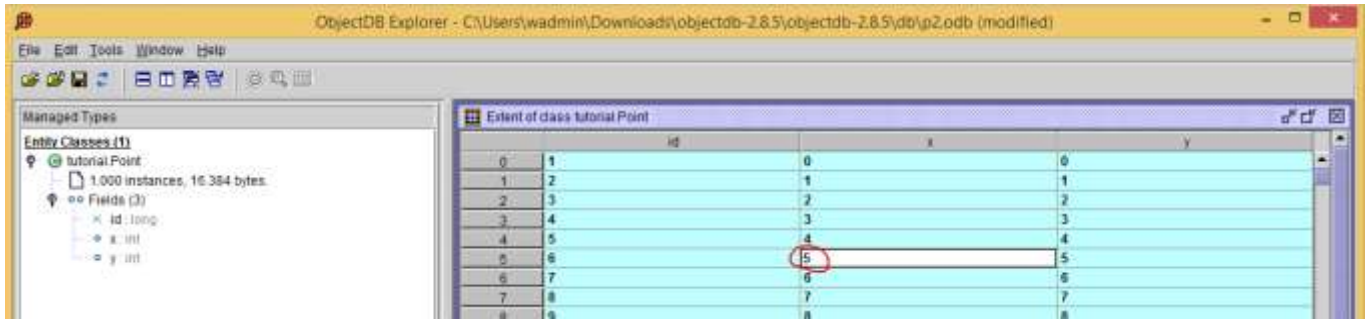
- ❖ Podemos ver el valor de los campos en tipo árbol, como si fuera un objeto, es decir, veremos la instancia almacenada en la base de datos:



- ❖ O en tipo tabla



Desde una vista u otra podemos hacer cambios en los valores de los campos, haciendo doble clic en campo:

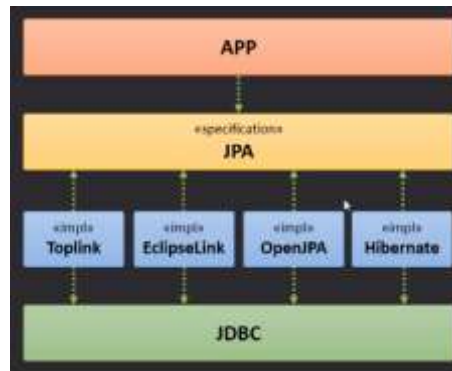


Os animo a explorar la base de datos 'world' para ver una estructura más compleja de una base orientada a objetos.

1.4 JPA vs JDBC

Y para terminar, la pregunta del millón, ¿JPA o JDBC?

✓ JPA Abstrae a JDBC



✓ Análisis de JPA

○ Ventajas de JPA:

- Nos permite desarrollar mucho más **rápido**.
- Trabajamos con **Entidades**.
- Elimina muchos **errores** en tiempo de ejecución.
- Legibilidad y **mantenimiento**.

○ Desventajas de JPA:

- No es tan **potente** como ejecutar queries nativos.
- Se **degrada** el performance, lo que supone que aumente el tiempo de respuesta.
- Requiere de una **curva** de aprendizaje.

✓ **Análisis de JDBC**

- **Ventajas de JDBC:**
 - Ofrece un **performance** superior a JPA.
 - Permite explotar al máximo las **características nativas** de la base de datos.
- **Desventajas de JDBC:**
 - El mantenimiento es mucho más **costoso**.
 - Introduce muchos **errores** en tiempo de ejecución.
 - El desarrollo es mucho más **lento**.

✓ **Conclusiones**

- Utilicemos JDBC cuando el **performance** es el factor más importante.
- Utilicemos JPA cuando busquemos **productividad** a costa del performance.

1.5 Referencias:

- ✓ ObjectDB: <https://es.wikipedia.org/wiki/ObjectDB>
- ✓ Quick start with NetBeans: <https://www.objectdb.com/tutorial/jpa/netbeans>
- ✓ Manual: <https://www.objectdb.com/java/jpa>
 - Using JPA: <https://www.objectdb.com/java/jpa/persistence>
 - JPA Queries: <https://www.objectdb.com/java/jpa/query>
 - Tools and Utilities -> Database Explorer: <https://www.objectdb.com/java/jpa/tool/explorer>