Acceso a Bases de Datos.

Caso práctico

Dantadd (CC BY)

Ada ha asignado un proyecto a **María** y a **Juan**. Se trata de un proyecto importante, y puede suponer muchas ventas, y por tanto una gran expansión para la empresa.

En concreto, un notario de renombre en el panorama nacional, se dirigió a BK programación para pedirles que les desarrolle un programa para su notaría, de modo que toda la gestión de la misma, incluyendo la emisión de las escrituras, se informatizaran. Además, si el programa es satisfactorio, se encargará de promocionar la aplicación ante el resto de sus conocidos notarios, pudiendo por tanto suponer muchas ventas y por ello, dinero.

Una cuestión vital en la aplicación es el almacenamiento de los datos. Los datos de los clientes, y de las escrituras deberán guardarse en bases de datos, para su tratamiento y recuperación las veces que haga falta.

Como en BK programación trabajan sobre todo con Java, desde el primer momento Juan y María tienen claro que van a tener que utilizar bases de datos relacionales y JDBC y así lo comentan con **Ada**.

¿Utilizarías un procesador de textos que no te da la opción de guardar el documento que estás editando? Como es obvio, a nadie se le ocurre hacer un programa así. De hecho, casi todos los programas hoy día tienen la opción de guardar los datos, sean o no procesadores de texto.

Hasta ahora, ya conoces como abrir un archivo y utilizarlo como “almacén” para los datos que maneja tu aplicación. Utilizar un archivo para almacenar datos es la forma más sencilla de persistencia, porque en definitiva, **la persistencia es hacer que los datos perduren en el tiempo**.

Hay muchas formas de hacer los datos de una aplicación persistentes, y muchos niveles de persistencia. Cuando los datos de la aplicación solo están disponibles mientras la aplicación se esta ejecutando, tenemos un nivel de persistencia muy bajo, y ese era el caso de Antonio: su aplicación no almacenaba los datos en ningún lado, y en posteriores ejecuciones los datos no podían ser utilizados, pues solo estaban disponibles mientras no cerraras la aplicación.

Lo deseable es que los datos de nuestra aplicación tengan un nivel de persistencia lo mayor posible. Tendremos un mayor nivel de persistencia si los datos “sobreviven” varias ejecuciones, o lo que es lo mismo, si nuestros datos se guardan y luego son reutilizables con posterioridad. Tendremos un nivel todavía mayor si “sobreviven” varias versiones de la aplicación, es decir, si guardo los datos con la versión 1.0 de la aplicación y luego puedo utilizarlos cuando este disponible la versión 2.0.

Pero lo verdaderamente interesante de esta unidad, es la forma de hacer persistentes los datos: la utilización de Bases de Datos. Utilizaremos bases de datos relacionales, que son las más utilizadas en la actualidad y haremos una introducción a las bases de datos orientadas a objetos.

1.- Introducción.

Hoy en día, la mayoría de aplicaciones informáticas necesitan almacenar y gestionar gran cantidad de datos.

Esos datos, se suelen guardar en bases de datos relacionales, ya que éstas son las más extendidas actualmente.

Las bases de datos relacionales permiten organizar los datos en tablas y esas tablas y datos se relacionan mediante campos clave. Además se trabaja con el lenguaje estándar conocido como SQL, para poder realizar las consultas que deseemos a la base de datos.

José Javier Bermúdez Hernández (CC BY-NC)

Una base de datos relacional se puede definir de una manera simple como aquella que presenta la información en tablas con filas y columnas.

Una **tabla es una serie de filas y columnas**, en la que **cada fila es un registro** y cada columna es un campo. Un **campo** representa un dato de los elementos almacenados en la tabla (NSS, nombre, etc.) Cada registro representa un elemento de la tabla (el equipo Real Madrid, el equipo Real Murcia, etc.)

No se permite que pueda aparecer dos o más veces el mismo registro, por lo que uno o más campos de la tabla forman lo que se conoce como clave primaria.

El sistema gestor de bases de datos, en inglés conocido como: **Database Management System (DBMS)**, gestiona el modo en que los datos se almacenan, mantienen y recuperan.

En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina: **Relational Database Management System (RDBMS)**.

Tradicionalmente, la programación de bases de datos ha sido como una Torre de Babel: gran cantidad de productos de bases de datos en el mercado, y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante JDBC (***Java Database Connectivity***), permite **simplificar el acceso a base de datos**, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

- Ser un API con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java,
- Aprovechar la experiencia de los APIs de bases de datos existentes,
- Ser sencillo.

Autoevaluación

JDBC permite acceder a bases de datos relacionales cuando programamos con Java, pudiendo así utilizar SQL.

○ Verdadero ○ Falso

Verdadero
JBDC sirve para acceder a bases de datos relacionales con Java.

Para saber más

Si necesitas refrescar o simplemente aprender el concepto de clave primaria, en la wikipedia puedes consultarlo.

Clave primaria.

Debes conocer

Si no has estudiado nunca bases de datos, ni tienes idea de qué es SQL o el modelo relacional, sería conveniente que te familiarizaras con él. A continuación te indicamos un tutorial bastante ameno sobre SQL y en donde describe brevemente el modelo relacional.

Tutorial SQL.

Otro recurso muy interesante para aprender o consultar claúsulas de SQL es el que tienes en el siguiente enlace de la web **w3schools**. Como ya sabes, contiene multitud de ejemplos que se pueden ejecutar online.

SQL Tutorial w3schools

1.2.- JDBC.

JDBC es un API Java que hace posible ejecutar sentencias SQL.

De JDBC podemos decir que:

- Consta de un **conjunto de clases e interfaces** escritas en Java.
- Proporciona un API estándar para desarrollar aplicaciones de bases de datos con un API Java pura.

Con JDBC, no hay que escribir un programa para acceder a una base de datos Access, otro programa distinto para acceder a una base de datos Oracle, etc., sino que **podemos escribir un único programa** con el API JDBC y el programa se encargará de enviar las sentencias SQL a la base de datos apropiada. Además, y como ya sabemos, una aplicación en Java puede ejecutarse en plataformas distintas.

En el desarrollo de JDBC, y debido a la confusión que hubo por la proliferación de API's propietarios de acceso a datos, Sun buscó los aspectos de éxito de un API de este tipo, ODBC (Open Database Connectivity).

ODBC se desarrolló con la idea de tener un estándar para el acceso a bases de datos en entorno Windows.

Aunque la industria ha aceptado ODBC como medio principal para acceso a bases de datos en Windows, ODBC no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

El nivel de abstracción al que trabaja JDBC es alto en comparación con ODBC, la intención de Sun fue que supusiera la base de partida para crear librerías de más alto nivel.

JDBC intenta ser tan simple como sea posible, pero proporcionando a los desarrolladores la máxima flexibilidad.

Autoevaluación

JDBC es la versión de ODBC para Linux.

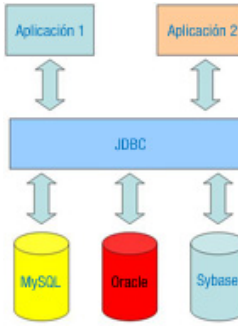
○ Verdadero ○ Falso

Falso
ODBC se desarrolló para ser un estándar para el acceso a bases de datos en entorno Windows.

1.3.- Conectores o Drivers.

El API JDBC viene distribuido en dos paquetes:

- **`java.sql`**, dentro de J2SE
- **`javax.sql`**, extensión dentro de J2EE



José Javier Bermúdez Hernández (CC BY-NC)

Un conector o driver es un conjunto de clases encargadas de implementar las interfaces del API y acceder a la base de datos.

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un conector adecuado. Un conector suele ser un fichero .jar que contiene una implementación de todas las interfaces del API JDBC.

Cuando se construye una aplicación de base de datos, **JDBC oculta los detalles específicos de cada base de datos**, de modo que le programador se ocupe sólo de su aplicación.

El conector lo proporciona el fabricante de la base de datos o bien un tercero.

El código de nuestra aplicación no depende del driver, puesto que trabajamos contra los paquetes**`java.sql`** y **`javax.sql`**.

JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos.
- Ejecutar una consulta.
- Procesar los resultados.

Ejemplo:

José Javier Bermúdez Hernández (CC BY-NC)

Código Java para establecer una conexión y ejecutar consulta (txt)

En principio, todos los conectores deben ser compatibles con ANSI SQL-2 Entry Level (ANSI SQL-2 se refiere a los estándares adoptados por el **American National Standards Institute** (ANSI) en 1992. Entry Level se refiere a una lista específica de capacidades de SQL). Los desarrolladores de conectores pueden establecer que sus conectores conocen estos estándares.

Para saber más

Lista sobre los conectores JDBC para acceder a muchas las bases de datos listadas (en inglés).

Conectores JDBC

1.4.- Instalación de la base de datos.

Lo primero que tenemos que hacer, para poder realizar consultas en una base de datos, es obviamente, instalar el sistema gestor de base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, no es imposible explicar la instalación de todas. Así que vamos a optar por una, en concreto por **MariaDB** ya que es un sistema gestor gratuito y que funciona en varias plataformas. Al igual que MySQL, es un servicio de manejo de bases de datos, cuenta con licencia GPL y de hecho fue creado por el desarrollador de MySQL, el conocido Monty Widenius, junto a un grupo de desarrolladores que decidieron formar parte del proyecto en forma voluntaria.

Para instalar MariaDB en Windows 10 puedes seguir los pasos que te detallamos en el siguiente video. Además muestra como durante el proceso podemos instalar también **HeidiSQL**, un cliente gráfico que nos permitirá conectarnos a nuestro servidor de bases de datos y llevar a cabo tareas de administración, entre ellas, crear nuevas bases de datos y tablas:

https://www.youtube.com/embed/7h8y-LEfMzA

Resumen textual alternativo

Si utilizas Linux, el sitio de descarga es el mismo que el indicado en la presentación anterior, y la instalación la puedes hacer con los pasos que te indican en el enlace:

Instalar MySQL/Maria DB en Linux.

Autoevaluación

Para usar programar accesos a MySQL hay que tener el driver JDBC para MySQL en nuestra aplicación.

○ Verdadero ○ Falso

Verdadero
De no ser así será imposible conectarnos a la base de datos.

Para saber más

Para probar con otras bases de datos, puedes instalar también Oracle. Aquí puedes ver como se instala.

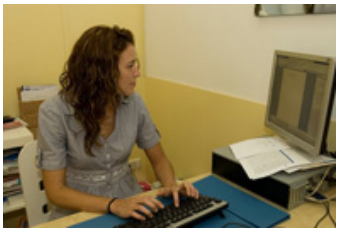
Instalar Oracle.

1.5.- Creación de las tablas en una base de datos.

Caso práctico

María, **Ada** y **Juan** han realizado concienzudamente el diseño de las tablas necesarias para la base de datos de la aplicación de notarías.

También se han decantado por el sistema gestor de bases de datos a utilizar. Emplearán un sistema gestor de bases de datos relacional. Una vez instalado el sistema gestor, tendrán que programar los accesos a la base de datos para guardar los datos, recuperarlos, realizar las consultas para los informes y documentos que sean necesarios, etc.



Ministerio de Educación y FP (CC BY-NC)

En Java podemos conectarnos y manipular bases de datos utilizando JDBC. Pero la creación en sí de la base de datos debemos hacerla con la herramienta específica para ello. Normalmente será el administrador de la base de datos, a través de las herramientas que proporcionan el sistema gestor, el que creará la base de datos. No todos los drivers JDBC soportan la creación de la base de datos mediante el lenguaje de definición de datos (**DDL**).

Normalmente, cualquier sistema gestor de bases de datos incluye asistentes gráficos para crear la base de datos con sus tablas, claves, y todo lo necesario.

También, como en el caso de MySQL, o de Oracle, y la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la línea de comandos de MySQL o de Oracle, con las sentencias SQL apropiadas.

Vamos utilizar el sistema gestor de base de datos MySQL, por ser un producto gratuito y de fácil uso, además de muy potente.

En el apartado anterior hemos podido comprobar en el video que crear bases de datos y administrarlas desde HeidiSQL es relativamente sencillo pero si quieres profundizar en ello, aquí tienes un videotutorial:

https://www.youtube.com/embed/MwcvpT27ugs

Resumen textual alternativo

Para saber más

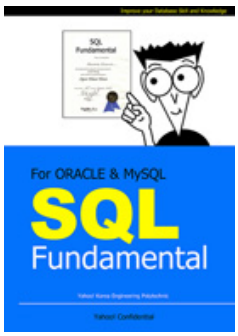
Otra posibilidad para la instalación del SGBD MariaDB y un cliente es utilizar paquetes integrados como XAMPP. XAMPP es una distribución de Apache completamente gratuita y fácil de instalar que contiene MariaDB, Apache, PHP y Perl. El paquete de instalación de XAMPP ha sido diseñado para ser increíblemente fácil de instalar y usar. Como cliente del SGBD utiliza una aplicación web denominada **phpmyadmin**, ampliamente conocida entre los desarrolladores.

Esta opción suele ser la elegida en el desarrollo web actual, dado que el paquete contiene el servidor web Apache.

XAMPP está disponible para diferentes sistemas operativos.

Web de XAMPP

2.- Lenguaje SQL.



Jinho Jung (CC BY-NC)

¿Cómo le pedimos al Sistema Gestor de Bases de Datos Relacional (SGBDR), en concreto en este caso, al de MySQL, que nos proporcione la información que nos interesa de la base de datos?

Se utiliza el **lenguaje SQL** (Structured Query Language) para interactuar con el SGBDR.

SQL es un lenguaje **no procedimental** en el cual se le indica al SGBDR **qué** queremos obtener y **no cómo hacerlo**. El SGBDR analiza nuestra orden y si es correcta sintácticamente la ejecuta.

El estudio de SQL nos llevaría mucho más que una unidad, y es objeto de estudio en otros módulos de este ciclo formativo. Pero como resulta imprescindible para poder continuar, haremos una mínima introducción sobre él.

Los comandos SQL se pueden dividir en dos grandes grupos:

- Los que se utilizan para **definir las estructuras de datos**, llamados comandos **DDL** (Data Definition Language).
- Los que se utilizan para **operar con los datos almacenados en las estructuras**, llamados **DML** (Data Manipulation Language).

En la siguiente presentación encontrarás algunos de los comandos SQL más utilizados.

Comandos SQL básicos

CREATE TABLE

- Descripción:

Se utiliza para **crear** una tabla especificando el nombre de la misma. Al crear la tabla es necesario especificar cada una de las columnas que la forman y el tipo de dato correspondiente.

- Sintaxis:

```
CREATE TABLE <tabla> (
<columna1> <tipo de dato>,
<columna2> <tipo de dato>,...,
<columnaN> <tipo de dato>)
```

Comandos SQL básicos II

CREATE TABLE

- Ejemplo de uso:

```
CREATE TABLE empleado (
  nombre CHAR(45),
  f_nacimiento DATE,
  departamento NUMBER(2),
  salario NUMBER(6,2) )
```

Comandos SQL básicos III

INSERT INTO

- Descripción:

Se utiliza para **añadir** registros (filas) a una tabla.

- Sintaxis:

```
INSERT INTO <tabla> VALUES (
<valor columna1>,
<valor columna2>, ...,
<valor columnaN>)
```

- Ejemplo de uso:

```
INSERT INTO empleado VALUES (
'José Gómez','10/02/1977',15,1200.00)
```

Comandos SQL básicos IV

DELETE

- **Descripción:**
Se utiliza para borrar registros (filas) de una tabla.

- **Sintaxis:**
DELETE FROM <tabla> WHERE <condición>

- **Ejemplo de uso:**
DELETE FROM empleado WHERE nombre = 'José Pérez')

Comandos SQL básicos V

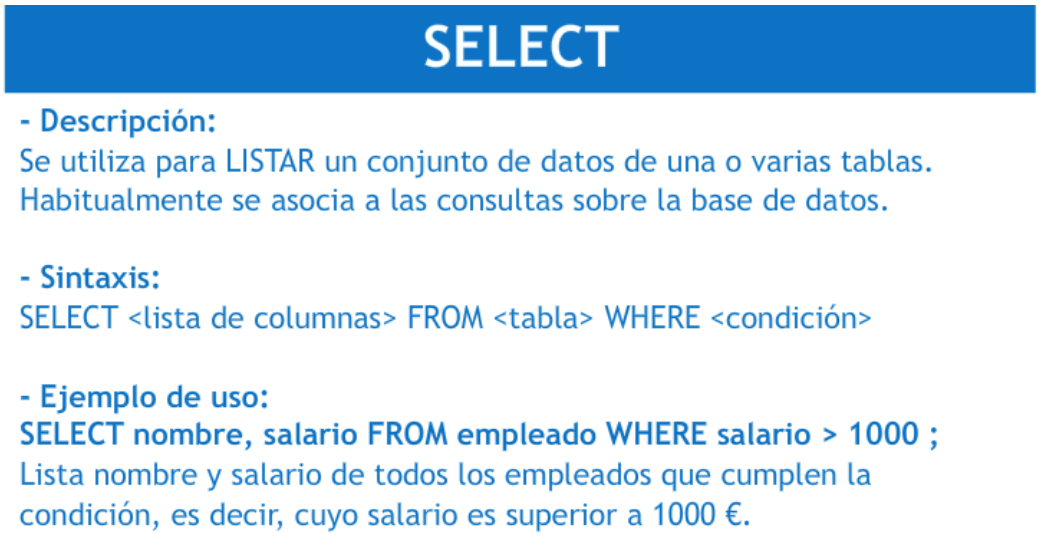
UPDATE

- **Descripción:**
Se utiliza para **modificar** datos de una tabla.

- **Sintaxis:**
UPDATE <tabla> SET <columna>=<expresión>... WHERE <condición>

- **Ejemplo de uso:**
UPDATE empleado
SET salario = salario+100
WHERE nombre = 'José Navarro' ;

Comandos SQL básicos VI



Comandos SQL básicos VII

Resumen textual alternativo

Para saber más

En este enlace encontrarás de una manera breve pero interesante la historia del SQL

Historia del SQL.

Autoevaluación

Con las consultas SQL hay que especificar paso a paso cómo recuperar cada dato de la base de datos.

○ Verdadero ○ Falso

Falso
Basta con decirle qué datos queremos y el sistema gestor se encarga de obtenerlos.

2.1.- Lenguaje SQL (II).

José Javier Bermúdez Hernández (CC BY-NC)

La primera fase del trabajo con cualquier base de datos comienza con sentencias DDL, puesto que antes de poder almacenar y recuperar información debemos definir las estructuras donde agrupar la información. Las estructuras básicas con las que trabaja SQL son las tablas.

Como hemos visto antes, una tabla es un conjunto de celdas agrupadas en filas y columnas donde se almacenan elementos de información.

Antes de llevar a cabo la creación de una tabla conviene planificar: nombre de la tabla, nombre de cada columna, tipo y tamaño de los datos almacenados en cada columna, información adicional, restricciones, etc.

Hay que tener en cuenta también ciertas restricciones en la formación de los nombres de las tablas: longitud. Normalmente, aunque dependen del sistema gestor, suele tener una longitud máxima de 30 caracteres, no puede haber nombres de tabla duplicados, deben comenzar con un carácter alfabético, permitir caracteres alfanuméricos y el guión bajo '_', y normalmente no se distingue entre mayúsculas y minúsculas.

Por ejemplo para crear una tabla de departamentos podríamos hacer:

```
CREATE TABLE departa (

   cod_dep number(3),

   nombre varchar2(15) not null,

   loc varchar2(10),

   constraint dep_pk primary key (cod_dep),

   constraint dep_loc check

            (loc in ('Madrid', 'Barcelona', 'Murcia'))

);
```

donde creamos la tabla con **`cod_dep`** como clave primaria. Además, se añade una restricción para comprobar que cuando se esté dando de alta un registro, lo que se escriba en el campo **`loc`** sea Madrid, Barcelona o Murcia.

Y una tabla de empleados, teniendo en cuenta el departamento en el que trabajen:

```
CREATE TABLE emp (

   cod_emp number(3),

   nombre varchar2(10) not null,

   oficio varchar2(11),

   jefe number(3),

   fecha_alta date,

   salario number(10),

   comision number(10),

   cod_dep number(3),

   constraint emp_pk primary key (cod_emp),

   constraint emp_fk foreign key (cod_dep) references departa(cod_dep)

            on delete cascade,

   constraint emp_ck check (salario > 0)

);
```

En el caso de esta tabla, se puede ver que hay un restricción para comprobar que el salario sea mayor que 0.

3.- Establecimiento de conexiones.

Caso práctico

Tanto **Juan** como **María** saben que trabajar con bases de datos relacionales en Java es tremendamente sencillo, por lo que establecer una conexión desde un programa en Java, a una base de datos, es muy fácil.

Juan le comenta a **María**: -Empleando la tecnología sólo necesitamos dos simples sentencias Java para conectar la aplicación a la base de datos. **María**, prepárate que en un periquete tengo lista la conexión con la base de datos y salimos a tomar un café.

Ministerio de Educación y FP (CC BY-NC)

Cuando queremos acceder a una base de datos para operar con ella, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos podemos utilizar el método **`getConnection()`** de la clase **`DriverManager`**. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión.

La ejecución de este método devuelve un objeto **`Connection`** que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, **`DriverManager`** itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún driver adecuado, se lanza una **`SQLException`**.

CC-by (CC BY-NC-SA)

Veamos un ejemplo comentado:

```
/**
 *
 * @author
 */
public class ConnectionDB {

    public static Connection openConnectDae() {
        Connection con =null;
        try {
        // Cargar el driver de mysql
            Class.forName("org.mariadb.jdbc.Driver");
        // Cadena de conexión para conectar con MySQL en localhost,
        //seleccionar la base de datos llamada BrancesionariaDB
        // con usuario y contraseña en servidor de MySQL: admin y admin
            String connectionUrl = "jdbc:mariadb://localhost:3306/concesionaria?user=admin&password=admin";
        // Obtener la conexión
            con= DriverManager.getConnection(connectionUrl);

        } catch (SQLException el ) {
            System.out.println("SQL Exception: " + e.toString());
        } catch (ClassNotFoundException cE) {
            System.out.println("Excepción: " + cE.toString());
        }
        return con;
    }
}
```

José Javier Bermúdez Hernández (CC BY-NC)

Si probamos este ejemplo con NetBeans, o cualquier otro entorno, y no hemos instalado el conector para MySQL, en la consola obtendremos el mensaje: Excepción: **`java.lang.ClassNotFoundException: com.mysql.jdbc.Driver.`**

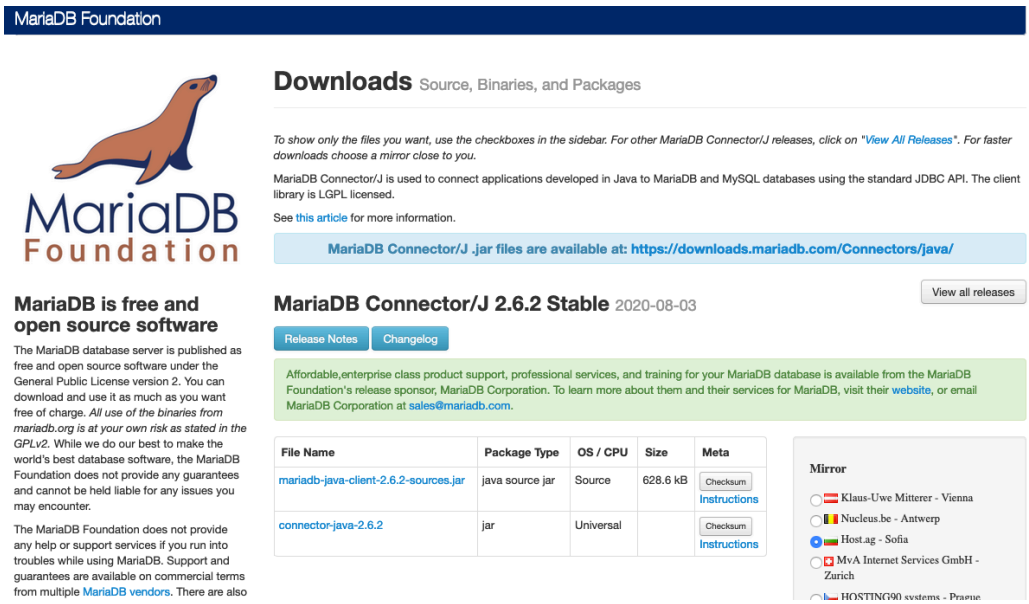
3.1.- Instalar el conector de la base de datos.

En la siguiente presentación vamos a ver cómo descargarnos el conector o driver que necesitamos para trabajar con MariaDB. Como verás, tan sólo consiste en descargar un archivo, descomprimirlo y desde NetBeans añadir el fichero **`.jar`** como librería a nuestro proyecto:

Instalar Driver MariaDB

El primer paso será descargar el driver desde la web. Normalmente se suministra en forma de fichero jar.

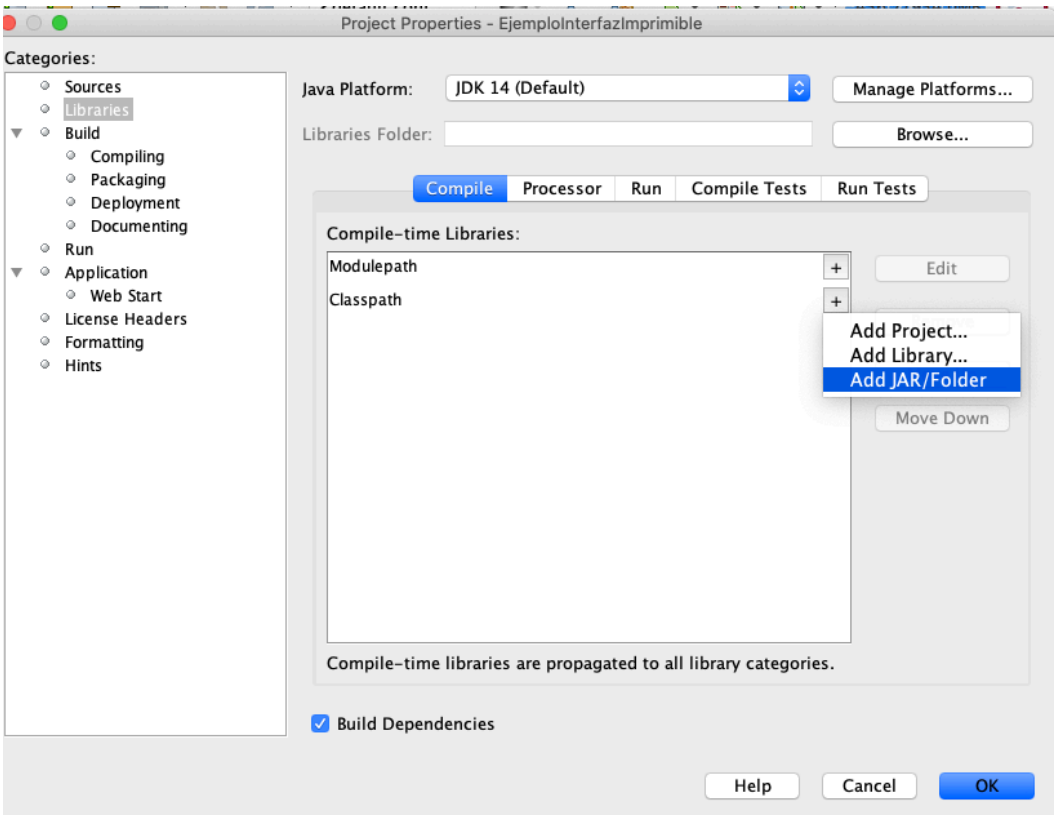
[Descarga MariaDB Connector]



Instalar Driver MariaDB II

Una vez descargado el fichero .jar, tan solo tendremos que añadirlo a todo proyecto que quiera establecer conexiones contra MariaDB. Para ello, realizamos los pasos ya realizados en otras ocasiones, por ejemplo, con las librerías de JavaFX.

1. Accede a las propiedades del proyecto (botón derecho sobre su nombre y pulsar en **Properties**).
2. Seleccionamos en el panel izquierdo la opción **Libraries** y agregarmos el driver en forma de fichero jar.
3. Tras finalizar, ya tenemos nuestro proyecto configurado para establecer conexiones con MariaDB.



Instalar Driver MariaDB III

Resumen textual alternativo

Por tanto, como ya hemos comentado anteriormente, entre el programa Java y el Sistema Gestor de la Base de Datos (SGBD) se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las cl de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

La función del conector es traducir los comandos del API JDBC al protocolo nativo del SGBD.

Autoevaluación

Para establecer una conexión con una base de datos se puede usar `getConnection()`.

○ Verdadero ○ Falso

Verdadero
En efecto getConnection se utiliza para conectarnos a una base de datos.

3.2.- Registrar el controlador JDBC.

Al fin y al cabo ya lo hemos visto en el ejemplo de código que poníamos antes, pero incidimos de nuevo. Registrar el controlador que queremos utilizar es tan fácil como escribir una línea de código.

Bruno Cordioli (CC BY)

Hay que consultar la documentación del controlador que vamos a utilizar para conocer el nombre de la clase que hay que emplear. En el caso del controlador para MySQL es "**`com.mysql.jdbc.Driver`**", o sea, que se trata de la clase **Driver** que está en el paquete **`com.mysql.jdbc`** del conector que hemos descargado, y que has observado que no es más que una librería empaquetada en un fichero .jar.

La línea de código necesaria en este caso, en la aplicación Java que estemos construyendo es:

```
// Cargar el driver de mysql


Class.forName("com.mariadb.jdbc.Driver");
```

Una vez cargado el controlador, es posible hacer una conexión al SGBD.

Hay que asegurarse de que si no utilizáramos NetBeans u otro IDE, para añadir el .jar como hemos visto, entonces el archivo .jar que contiene el controlador JDBC para el SGBD habría que incluirlo en el CLASSPATH que emplea nuestra máquina virtual, o bien en el directorio ext del JRE de nuestra instalación del JDK.

Hay una excepción en la que no hace falta ni hacer eso: en caso de utilizar un acceso mediante puente JDBC-ODBC, ya que ese driver está incorporado dentro de la distribución de Java, por lo que no es necesario incorporarlo explícitamente en el classpath de una aplicación Java. Por ejemplo, sería el caso de acceder a una base de datos Microsoft Access.

4.- Ejecución de consultas sobre la base de datos.

Caso práctico

Ada está echando una mano a **Juan** y **María** en la creación de consultas, para los informes que la aplicación de notaría debe aportar a los usuarios

de la misma.

Hacer consultas es una de las facetas de la programación que más entretiene a **Ada**, le resulta muy ameno y fácil. Además, y dada la importancia del proyecto, cuanto antes avancen en él, mucho mejor.

Por suerte, los tres: Ada, María y Juan tienen experiencia en consultas SQL y saben que, cuando se hace una consulta a una base de datos, hay que afinar y hacerla lo más eficiente posible, pues si se descuidan el sistema gestor puede tardar mucho en devolver los resultados. Además, algunas consultas pueden devolver un conjunto de registros bastante grande, que puede resultar difícil de manejar desde el programa, ya que por norma general tendremos que manejar esos datos registro a registro.

Para operar con una base de datos ejecutando las consultas necesarias, nuestra aplicación deberá hacer las operaciones siguientes:

- **Cargar el conector** necesario para comprender el protocolo que usa la base de datos en cuestión.
- **Establecer una conexión** con la base de datos.
- **Enviar consultas** SQL y procesar el resultado.
- **Liberar los recursos** al terminar.
- **Gestionar los errores** que se puedan producir.

Podemos utilizar los siguientes tipos de sentencias:

- `Statement`: para sentencias sencillas en SQL.
- `PreparedStatement`: para consultas preparadas, como por ejemplo las que tienen parámetros.
- `CallableStatement`: para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

- **Consultas**: `SELECT`. Para las sentencias de consulta que obtienen datos de la base de datos, se emplea el método `ResultSet executeQuery(String sql)`. El método de ejecución del comando SQL devuelve un objeto de tipo `ResultSet` que sirve para contener el resultado del comando SELECT, y que nos permitirá su procesamiento.
- **Actualizaciones**: `INSERT, UPDATE, DELETE`, sentencias DDL. Para estas sentencias se utiliza el método `executeUpdate(String sql)`

Autoevaluación

Para poder enviar consultas a la base de datos hemos tenido que conectarnos a ella previamente.

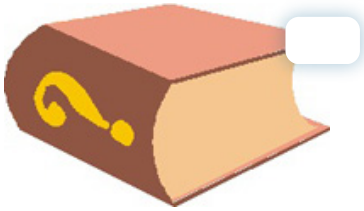
○ Verdadero ○ Falso

Verdadero
Así es, hay que cargar el driver y establecer la conexión.

4.1.- Recuperación de información (I).

Las consultas a la base de datos se realizan con sentencias SQL que van **"embebidas"** en otras sentencias especiales que son propias de Java. Por tanto, podemos decir que las consultas SQL las escribimos como parámetros de algunos métodos Java que reciben el `String` con el texto de la consulta SQL.

Las consultas devuelven un **ResultSet**, que es una clase java parecida a una lista en la que se aloja el resultado de la consulta. Cada elemento de la lista es uno de los registros de la base de datos que cumple con los requisitos de la consulta.



El ResultSet no contiene todos los datos, sino que los va obteniendo de la base de datos según se van pidiendo. La razón de esto es evitar que una consulta que devuelva una cantidad muy elevada de registros, tarde mucho tiempo en obtenerse y sature la memoria del programa.

Con el **ResultSet** hay disponibles una serie de métodos que permiten movernos hacia delante y hacia atrás en las filas, y obtener la información de cada fila.

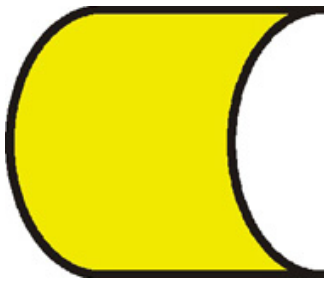
Por ejemplo, para obtener: nif, nombre, apellidos y teléfono de los clientes que están almacenados en la tabla del mismo nombre, de la base de datos *notarbd* que se creó anteriormente, haríamos la siguiente consulta:

```
// Preparamos la consulta y la ejecutamos

Statement s = n.createStatement();

ResultSet rs = s.executeQuery ("SELECT NIF, NOMBRE,"

      + "APELLIDOS, TELÉFONO FROM CLIENTE");
```

El método **next()** del **ResultSet** hace que dicho puntero avance al siguiente registro. Si lo consigue, el método **next()** devuelve **true**. Si no lo consigue, porque no haya más registros que leer, entonces devuelve **false**.

4.1.1.- Recuperación de información (II).

El método **executeQuery** devuelve un objeto **ResultSet** para poder recorrer el resultado de la consulta utilizando un cursor.



Ministerio de Educación y FP (CC BY-NC)

Para obtener una columna del registro utilizamos los métodos **get**. Hay un método **get**... para cada tipo básico Java y para las cadenas.

Un método interesante es **wasNull** que nos informa si el último valor leído con un método **get** es nulo.

Cuando trabajamos con el **ResultSet**, en cada registro, los métodos **getInt(), getString(), getDate()**, etc., nos devuelve los valores de los campos de dicho registro. Podemos pasar a estos métodos un índice (que comienza en 1) para indicar qué columna de la tabla de base de datos deseamos, o bien, podemos usar un **String** con el nombre de la columna (tal cual está en la tabla de base de datos).

José Javier Bermúdez Hernández (CC BY-NC)

Código consulta e iteración.

Autoevaluación

Para obtener un entero almacenado en uno de los campos de un registro, trabajando con el `ResulSet` emplearemos el método `getInt()`.

○ Verdadero ○ Falso

Verdadero
Eso es. Hay un `getString()` para cadenas, `getDate()` para fechas, etc.

4.2.- Actualización de información.

José Javier Bermúdez Hernández. (CC BY-NC)

Respecto a las consultas de actualización, **`executeUpdate`**, retornan el número de registros insertados, registros actualizados o eliminados, dependiendo del tipo de consulta que se trate.

Supongamos que tenemos varios registros en la tabla **Cliente**, de la base de datos notarbd con la que seguimos trabajando. Si quisiéramos actualizar el teléfono del tercer registro, que tiene **idCLIENTE=3** y ponerle como nuevo teléfono el **968610009** tendríamos que hacer:

```
String connectionUrl = "jdbc:mysql://localhost/notarbd?" +

                                    "user=root&password=admin";

// Obtener la conexión

Connection con = DriverManager.getConnection(connectionUrl);

// Preparamos la consulta y la ejecutamos

Statement s = con.createStatement();

s.executeUpdate("UPDATE CLIENTE SET teléfono='968610009' WHERE idCLIENTE=3");

// Cerramos la conexión a la base de datos.

con.close();
```

4.3.- Adición de información.

Darwin Bell (CC BY-NC)

Si queremos añadir un registro a la tabla Cliente, de la base de datos con la que estamos trabajando tendremos que utilizar la sentencia **`INSERT INTO`** de SQL. Al igual que hemos visto en el apartado anterior, utilizare **`executeUpdate`** pasándole como parámetro la consulta, de inserción en este caso.

Así, un ejemplo sería:

```
// Preparamos la consulta y la ejecutamos
Statement s = con.createStatement();
s.executeUpdate( "INSERT INTO CLIENTE" +
" (idCLIENTE, NIF, NOMBRE, APELLIDOS, DIRECCIÓN, CPOSTAL, TELÉFONO, CORREOELEC)"  +
" VALUES (4, '66778998T', 'Alfredo', 'Gates Gates', 'C/ Pirata 23','20400', '891222112', 'prueba@eresmas.es' )") ;
```

Autoevaluación

Al añadir registros a una tabla de una base de datos, tenemos que pasar como parámetro al **`executeUpdate()`**, una sentencia SQL del tipo: **`DELETE`**...

○ Verdadero ○ Falso

Falso
Se le debe pasar una sentencia de tipo **`INSERT INTO`**.

4.4. Borrado de información.

Cuando nos interese eliminar registros de una tabla de una base de datos, emplearemos la sentencia SQL: **`DELETE`**. Así, por ejemplo, si queremos eliminar el registro a la tabla Cliente, de nuestra base de datos y correspondiente a la persona que tiene el nif: 66778998T, tendremos que utilizar el código siguiente.

```
// Preparamos la consulta y la ejecutamos

  Statement s = con.createStatement();

  numReg = res.executeUpdate( "DELETE FROM CLIENTE WHERE NIF= '66778998T' " );


// Informamos del número de registros borrados

  System.out.println ("\nSe borró " + numReg + " registro\n") ;
```

Autoevaluación

Al ejecutar el borrado de un registro mediante `executeUpdate(…)`, no podemos saber si el borrado eliminó alguna fila o no.

○ Verdadero ○ Falso

Falso
Efectivamente podemos saberlo porque se devuelve el número de registros borrados, como hemos visto en el código del ejemplo anterior.

4.5.- Cierre de conexiones.

Las conexiones a una base de datos consumen muchos recursos en el sistema gestor por ende en el sistema informático en general. Por ello, conviene cerrarlas con el método **`close()`** siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (**`Statement`** y **`PreparedStatement`**) y los resultados (**`ResultSet`**) para liberar los recursos.

Citas para pensar

Noble cosa es, aun para un anciano, el aprender.

Sófocles.

Para saber más

En el siguiente enlace puedes ver cómo se realiza una gestión de la conexión con Oracle, desde el registro del conector hasta el cierre de la misma. También se comenta sobre MySQL.

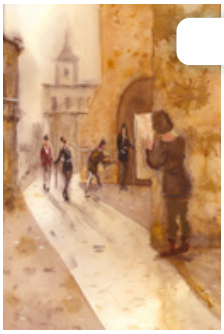
Gestión de la conexión en Oracle.

4.6.- Excepciones.

En todas las aplicaciones en general, y por tanto en las que acceden a bases de datos en particular, nos puede ocurrir con frecuencia que la aplicación no funciona, no muestra los datos de la base de datos que deseábamos, etc.

Es importante capturar las excepciones que puedan ocurrir para que el programa no aborte de manera abrupta. Además, es conveniente tratarlas para que nos den información sobre si el problema es que se está intentando acceder a una base de datos que no existe, o que el servicio MySQL no está arrancado, o que se ha intentado hacer

alguna operación no permitida sobre la base de datos, como acceder con un usuario y contraseña no registrados, ...



Por tanto es conveniente emplear el método `getMessage()` de la clase `SQLException` para recoger y mostrar el mensaje de error que ha generado MySQL, lo que seguramente nos proporcionará una información más ajustada sobre lo que está fallando.

Cuando se produce un error se lanza una excepción del tipo `java.sql.SQLException`.

- Es importante que **las operaciones de acceso a base de datos** estén **dentro de un bloque** `try-catch` que gestione las excepciones.
- Los objetos del tipo `SQLException` tienen dos métodos muy útiles para obtener el código del error producido y el mensaje descriptivo del mismo, `getErrorCode()` y `getMessage()` respectivamente.

El método `getMessage()` imprime el mensaje de error asociado a la excepción que se ha producido, que aunque esté en inglés, nos ayuda a saber qué ha generado el error que causó la excepción. El método `getErrorCode()`, devuelve un número entero que representa el código de error asociado. Habrá que consultar en la documentación para averiguar su significado.

Autoevaluación

El cierre de las conexiones y la gestión de excepciones sólo hay que efectuarla con bases de datos MySQL.

○ Verdadero ○ Falso

Falso
Siempre hay que gestionarlos con cualquier base de datos.

5.- El desface Objeto-Relacional

El desfase **objeto-relacional**, también conocido como **impedancia objeto-relacional**, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos. Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación**. El programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos**: en las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, que suelen ser sencillos, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.
- **Paradigma de programación**. En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

El modelo relacional trata con relaciones y conjuntos debido a su **naturaleza matemática**. Sin embargo, el modelo de Programación Orientada a Objetos trata con objetos y las asociaciones entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio

La escritura (y de manera similar la lectura) mediante JDBC implica: abrir una conexión, crear una sentencia en SQL y copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto. Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

Este problema es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en con u lenguajes de Programación Orientada a Objetos.

Podemos poner como ejemplo de desfase objeto-relacional, un Equipo de fútbol, que tenga un atributo que sea una colección de objetos de la clase Jugador. Cada jugador tiene un atributo "teléfono". Al transformar éste a relacional se ocuparía más de una tabla para almacenar la información, implicando varias sentencias SQL y bastante código.

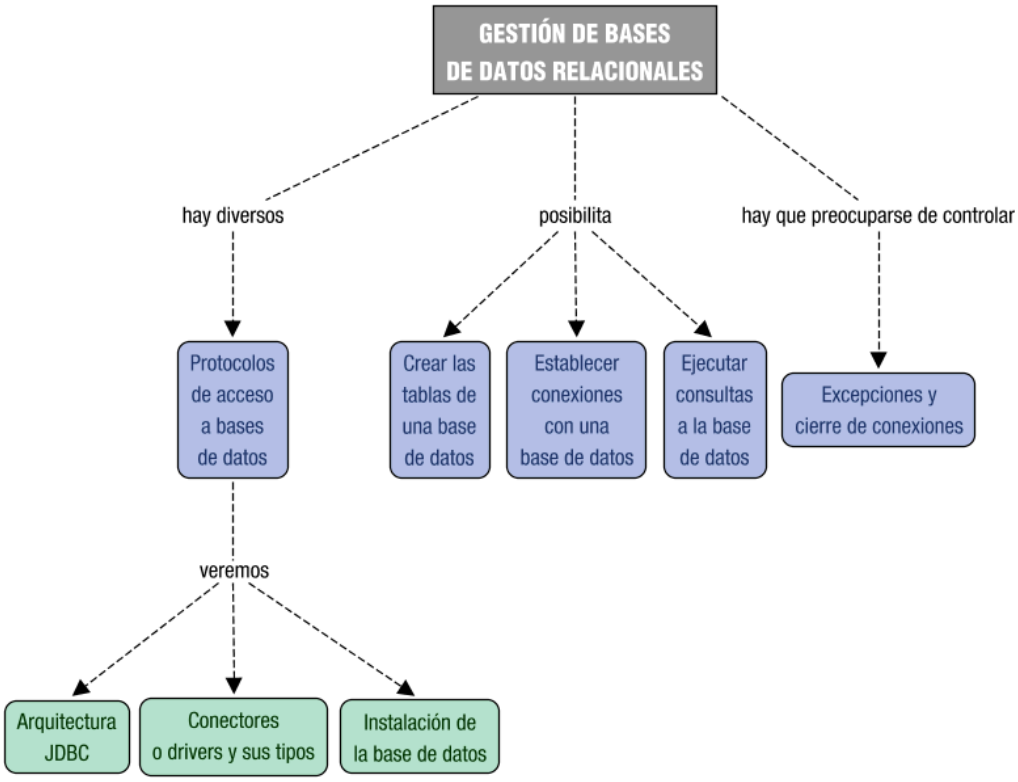
Las tendencias en la actualidad para rellenar el hueco existente entre un modelo relacional de almacenamiento y un modelo basado en objetos pasan por dos soluciones.

1. **Mapeadores Objeto-Relacional (ORM):** ORM es una técnica que persiste de forma transparente los objetos de aplicación a las tablas en una base de datos relacional. Estas herramientas se encargan de convertir objetos de la capa de negocio a tablas en el modelo relacional de forma transparente al programador. Como ventajas se puede hablar de: mas productividad trabajando con un modelo estrictramente basado en objetos en la capa de negocio y posibilidad de usar SGBDR que tan eficientes son actualmente. Como desventaja podemos hablar de mas necesidad de procesamiento por añadir una capa mas, la que se encarga de hacer la traducción de objetos. Ejemplos de ORM son Hibernate y JPA.
2. **Bases de Datos Orientadas a Objetos (SGBDOO):** son bases de datos constituidas por objetos de distintos tipos, sobre los que se definen una serie de operaciones para su interactuación, que a su vez se integran con las operaciones de un lenguaje de programación orientado a objetos (POO). Como ventajas podemos hablar de: ocultan información de objetos y sobre todo que es totalmente compatible con la capa de negocio. Como desventajas se puede hablar de que no existe un modelo teórico que los sustente y tampoco hay un modelo universalmente aceptado por todos los SGBD. Una herramienta que se puede probar porque dispone de versión gratuita es Intersystem Caché.

Estas herramientas serán trabajadas en profundidad en módulos profesionales de 2º curso.

6.- Conclusiones

Hemos finalizado el módulo "Programación" trabajando con bases de datos relacionales. El uso de bases de datos es fundamental en las aplicaciones sotfware actuales, tanto de escritorio como web. Las bases de datos relacionales son las mas utilizadas en la actualidad, aunque poco a poco están ganando terreno las bases de datos orientadas a objetos, que tratan de salvar el desfase objeto-relacional. JDBC es el API Java de alto nivel para el acceso a bases de datos relacionales.



Finalizada esta unidad, se puede decir que hemos adquirido los conocimientos base necesarios para desarrollar aplicaciones en Java. Estos conocimientos nos aportan las habilidades necesarias para afrontar el desarrollo de un proyecto o formar parte de un grupo de desarrollo. ¿Lo sabemos todo?. Por supuesto que no, de hecho ningún programador conoce toda la funcionalidad de un lenguaje. Pero si tenemos la base suficiente para

documentarnos frente a cualquier necesidad que nos pueda surgir. Y además, podremos aprender la sintaxis de otro lenguaje, puesto que los conocimientos de algoritmia y POO ya los tenemos.