UNIDAD 11: INTERACCIÓN CON BASES DE DATOS

Profesor: José Ramón Simó Martínez

Contenido

1. Introducción

En anteriores unidades hemos estudiado cómo almacenar información, de forma persistente, en nuestros programas a través de los ficheros. Sin embargo, cuando dicha información se compone de estructuras más amplias y complejas, la utilización de ficheros resulta ineficiente e insegura. Para ello, por tanto, necesitaremos de sistemas como las bases de datos.

En esta unidad aprenderemos a crear programas que interactúen con una base de datos a través de la API de JDBC.

Al terminar esta unidad deberás ser capaz de:

- Instalar un servicio básico de sistema gestor de base de datos MySQL.
- Configurar el driver que permite la interacción de un programa Java con la base de datos.
- Conocer la API JDBC y sus métodos principales para interactuar con una base de datos.
- Identificar las consultas parametrizadas y conocer sus ventajas
- Desarrollar programas que permitan almacenar y recuperar información en bases de datos.

Nota

En esta unidad se presuponen unos conocimientos mínimos de SQL. Para repasar conceptos básicos se deja el siguiente enlace:

https://www.w3schools.com/sql

2. Preparación del entorno de trabajo

La conexión de Java con una Base de Datos (BD) se realiza con dos herramientas básicas:

- **Gestor de base de datos (SGBD):** en esta unidad aprenderemos a conectarnos al SGBD de **MySQL**. Un SGBD es un software que permite gestionar una base de datos.
- **Driver de conexión al SGBD:** en nuestro caso utilizaremos el más popular conocido como **JDBC** (Java Database Connectivity). Básicamente, es una API (conjunto de clases y métodos) que permite la ejecución de operaciones sobre bases de datos desde Java.

La instalación del *SGBD MySQL* la realizaremos a partir del software **XAMPP** ya que es mucho más sencillo de configurar que otros entornos como *MySQL Workbench*, y para el propósito de esta unidad es suficiente.

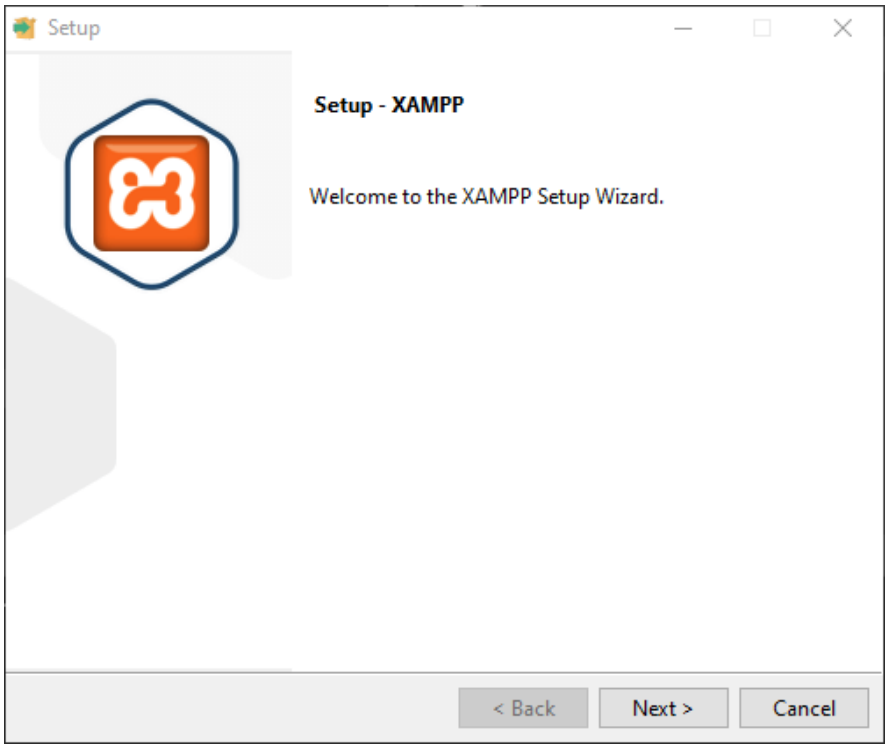
2.1. Instalación de XAMPP

XAMPP es un paquete de software libre (distribución de Apache) que permite, entre otras cosas, la gestión de una base de datos MySQL.

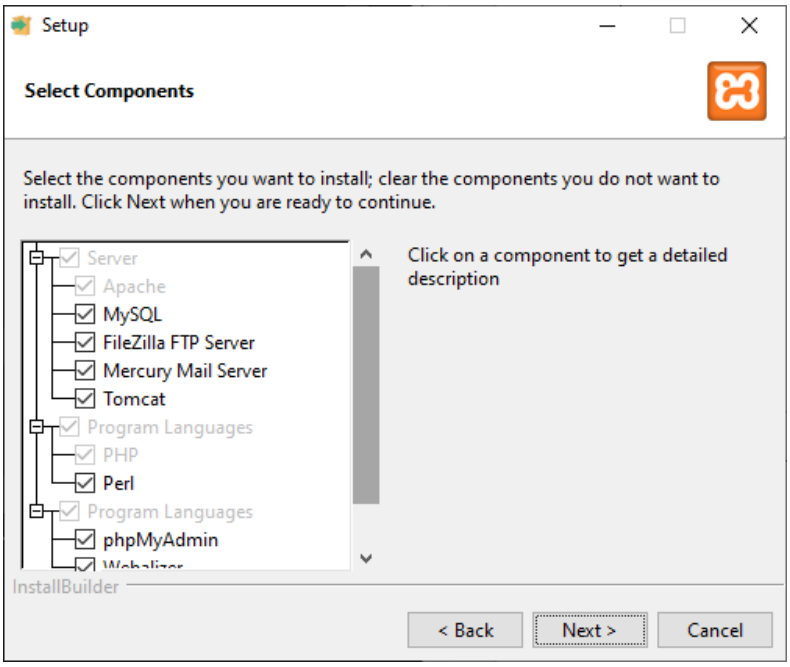
Para instalar XAMPP primero deberemos descargarnos la aplicación desde su página oficial (la última versión es la 8.2.4 en abril de 2023):

https://www.apachefriends.org/es/index.html

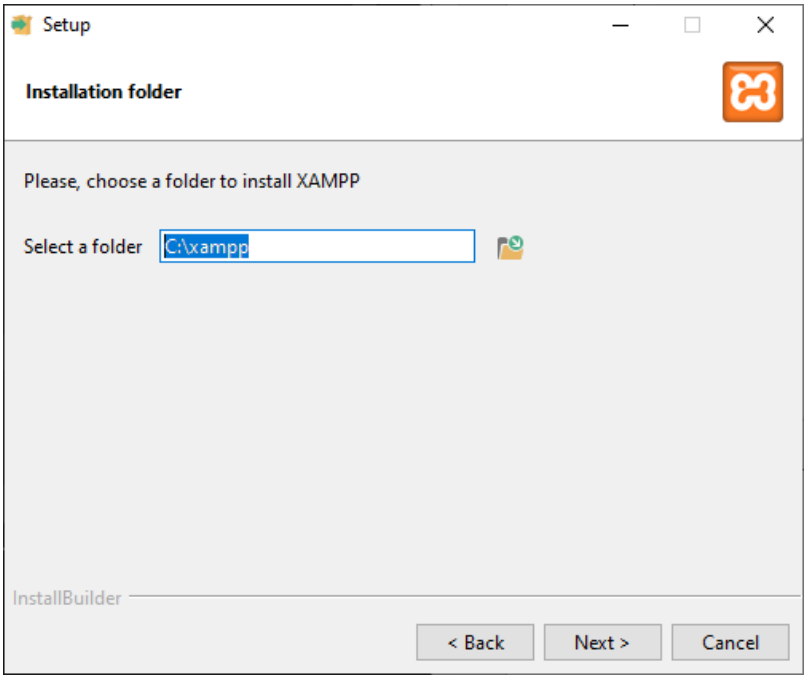
Ejecutaremos el software descargado y seguiremos los pasos que nos indica el proceso de instalación. A continuación, una breve explicación de los pasos a seguir:



En siguiente apartado podemos dejar todo seleccionado, aunque sería suficiente con tener activado MySQL, PHP y phpMyAdmin para nuestro propósito.

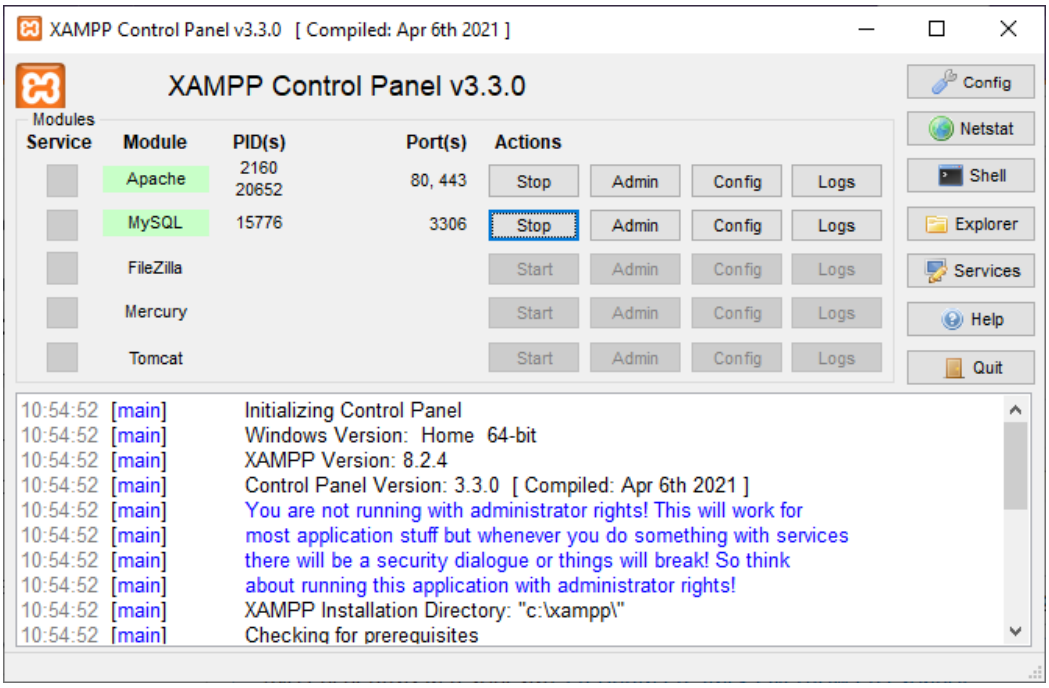


Es conveniente para la realización de pruebas que instalemos XAMPP en una carpeta que no tenga ningún problema con los permisos. En este caso, en c:\xampp (por defecto) sería lo recomendado:

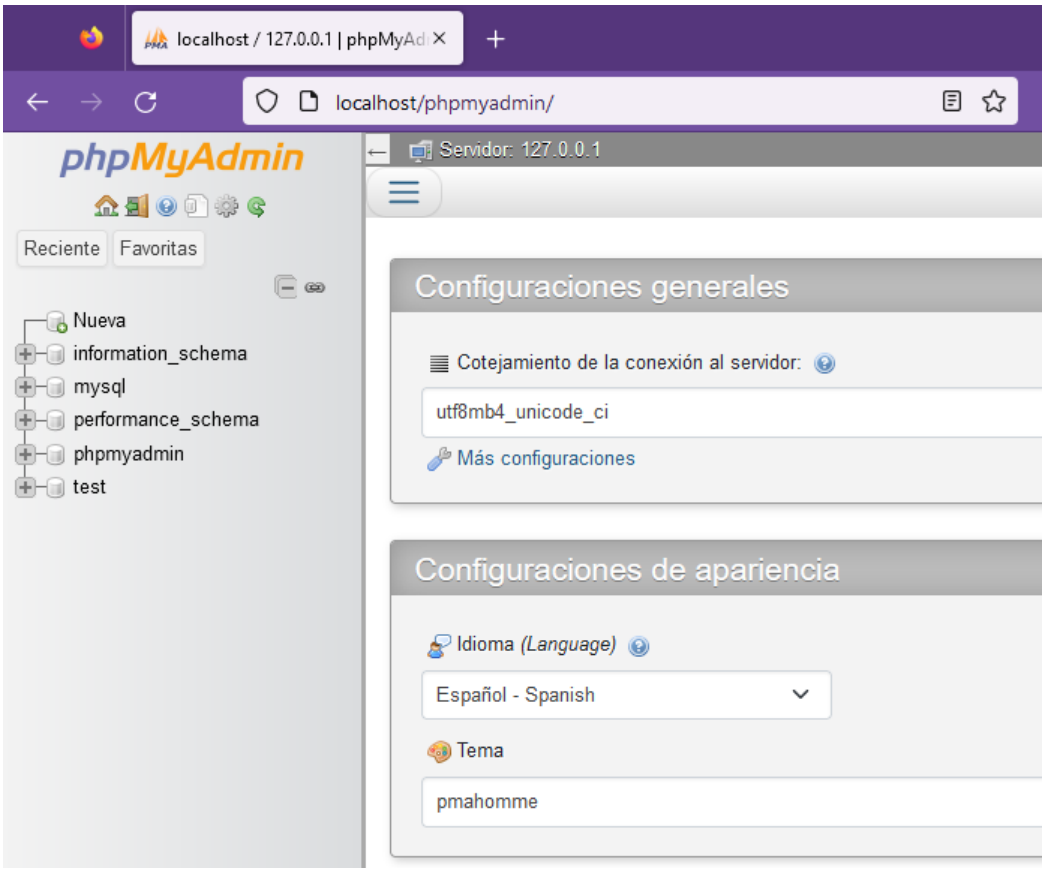


Ahora ya tendremos instalado el programa y dejaremos seleccionada la opción de ejecutarlo al finalizar la instalación.

Al iniciar el programa deberemos ejecutar (Start) los servicios de Apache y MySQL:



Finalmente, ejecutaremos el software de *phpMyAdmin* para la gestión de la base de datos desde el navegador web. Para ello, iremos a la barra de direcciones del navegador y escribiremos: *localhost/phpmyadmin*



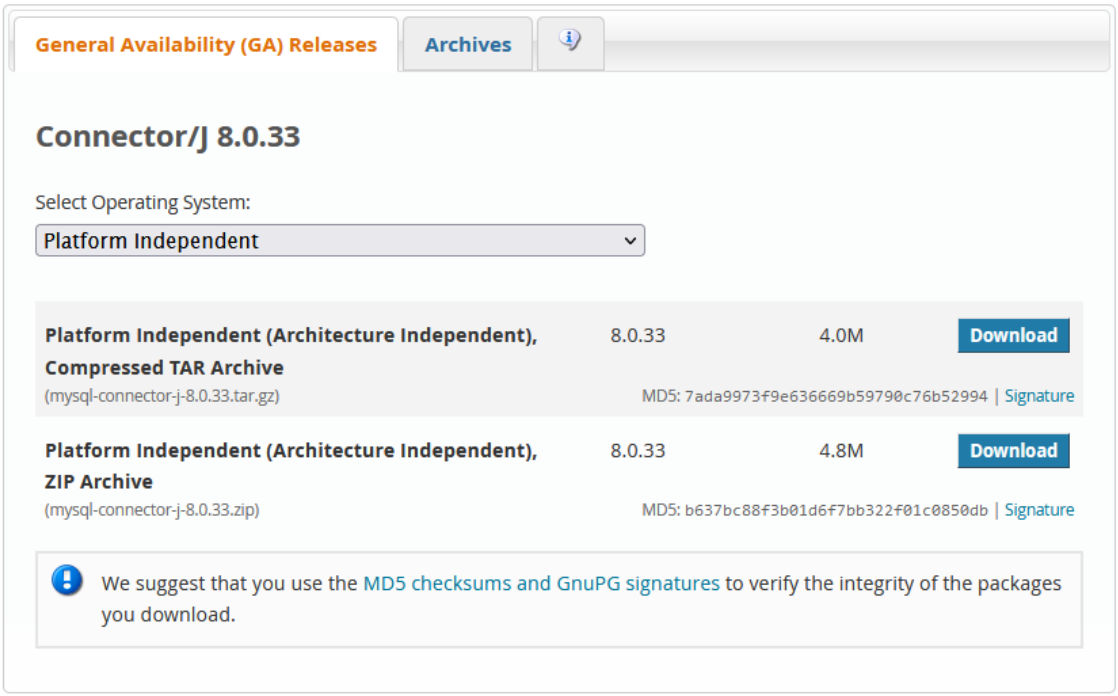
2.2. Driver de MySQL para Java y Eclipse

Ahora instalaremos el *driver* que permitirá conectar nuestra aplicación Java con la base de dato, ya que el JDK de Java no incluye todos los drivers para conectarse con los distintos SGBD que existen. Luego, conectaremos dicho driver con un proyecto Java en el entorno de desarrollo Eclipse.

Descarga del driver JDBC (Connector/J)

JDBC es la API que permite la conexión de una aplicación en Java con una base de datos. El *driver* oficial de MySQL se conoce como MySQL Connector/J. Para conseguir dicho *driver* debemos descargarlo desde la página web oficial de MySQL: https://dev.mysql.com/downloads/connector/j/

Seleccionamos el sistema operativo *Platform Independent* y descargamos el paquete de opción *ZIP Archive*:



Una vez descargado el paquete deberemos descomprimirlo en C: con el mismo nombre de carpeta:

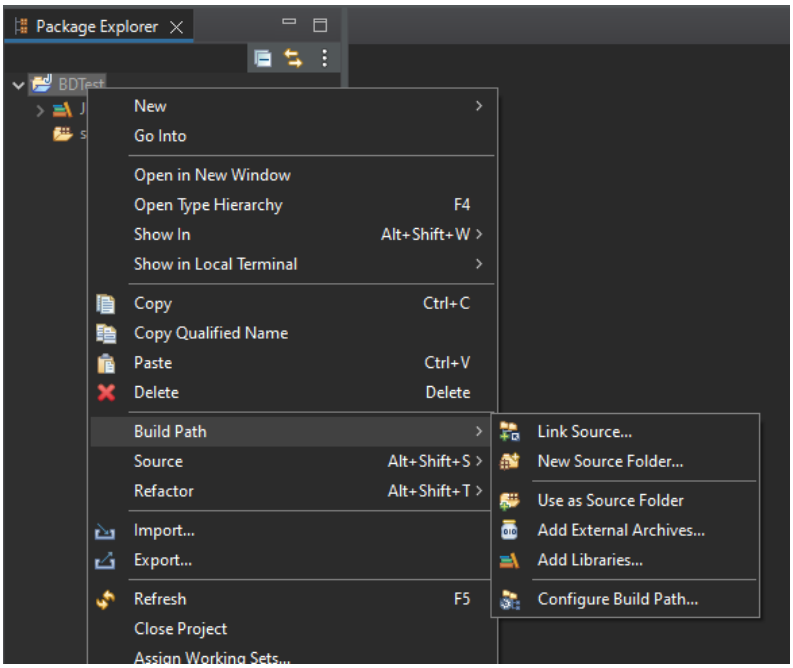
c:\mysql-connector-j-8.0.33

En dicha carpeta ahora encontraremos el *driver* (fichero *mysql-connector-j-8.0.33.jar*) que hace posible la conexión entre Java y una base de datos.

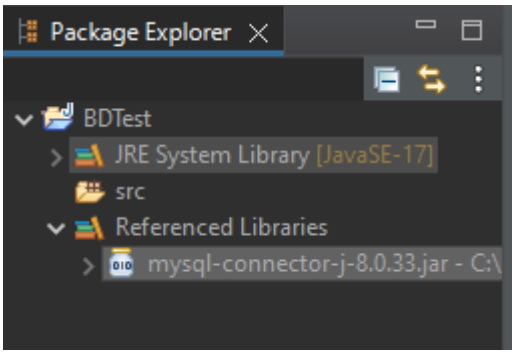
Agregar JDBC a Eclipse

El siguiente paso es agregar este *JDBC* en nuestro proyecto Java desde el entorno de desarrollo Eclipse. En primer lugar, deberemos crear un proyecto Java llamado por ejemplo *TestDB*.

A continuación, haremos clic derecho sobre el proyecto *-> Build Path -> Add External Archives…*



Buscamos el *driver* que hemos instalado anteriormente (fichero *mysql-connector-j-8.0.33.jar*) en *c:\mysql-connector-j-8.0.33*, y lo agregamos:



De esta manera ya tenemos el *driver* disponible en nuestro proyecto y listo para utilizarse. En el siguiente apartado estudiaremos las operaciones básicas de la API de JDBC para interactuar con una base de datos que hayamos creado.

3. Uso del *driver* JDBC en Java

Antes de estudiar este punto deberemos realizar los apartados anteriores. Asimismo, una vez realizadas las configuraciones previas necesarias, vamos a utilizar una base de datos desde una aplicación Java con el entorno de desarrollo Eclipse.

3.1. Preparativos

Previamente a la lógica de nuestro programa debemos tener en cuenta una serie de configuraciones en nuestro código.

Importando clases necesarias

Al principio de nuestro programa importaremos los siguientes paquetes:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.CallableStatement;
```

Las principales clases importadas son:

- *DriverManager*: cargará un Driver.
- *Connection*: establecerá conexiones con las bases de datos.
- *Statement*: ejecutará sentencias SQL.
- *PreparedStatement*: preparará la sentencia SQL (consultas parametrizadas)
- *ResultSet*: guardará el resultado de la consulta SQL.

Cargando del *driver*

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Con este método podremos usar un fichero de configuración externo para dar soporte al driver correspondiente y sus parámetros para conectarnos a la base de datos correspondiente. Cada SGBD necesitará cargar el *driver* cambiando el nombre de la clase, por ejemplo, para el SGBD Oracle:

```
Class.forName("oracle.jdbc.Driver.OracleDriver");
```

Estableciendo conexión

Una vez cargadas las clases necesarias y el driver, ya podemos establecer una conexión con la base de datos deseada:

```
Connection conn = null;

String usuario = "root";
String contrasenya = "";
String url = "jdbc:mysql://localhost/bdprueba"; // bdprueba es el nombre de la BD

conn = DriverManager.getConnection(url, usuario, contrasenya);
```

También podría hacerse separando de la URL el nombre de la base de datos:

```
Connection conn = null;

String nombreBD = "bdprueba";
String usuario = "root";
String contrasenya = "";
String url = "jdbc:mysql://localhost/" + nombreBD;

conn = DriverManager.getConnection(url, usuario, contrasenya);
```

3.2. Operaciones básicas sobre una BD

Para interactuar con una base de datos, es necesario seguir tres pasos fundamentales:

- Definir la sentencia SQL que se desea ejecutar.
- Ejecutar la sentencia y obtener los resultados deseados (si los hay).
- Cerrar conexiones.

Dependiendo del tipo de sentencia SQL que se ejecute, se puede obtener una respuesta que contenga filas o no recibir ningún tipo de respuesta. En el caso de una sentencia *Select*, se obtendrán filas como respuesta; en el resto de las sentencias no se devuelve nada.

En JDBC, hay dos formas de ejecutar una consulta SQL en una base de datos:

- Sentencia directa.
- Sentencia preparada.

Una *sentencia directa* (**Statement**) es una consulta SQL que se envía a la base de datos tal cual está escrita:

```
String sentencia = "SELECT …";
Statement s = conn.createStatement();
ResultSet rs = s.executeQuery(sentencia);
```

Por otro lado, una *sentencia preparada* (**Prepared Statement**) es una consulta SQL que se prepara previamente antes de enviarla a la base de datos:

```
String sentencia = "SELECT …";
PreparedStatement ps = conn.prepareStatement(sentencia);
ResultSet rs = ps.executeQuery(sentencia);
```

Luego, recorreremos las filas obtenidas y los campos de cada una de estas:

```
// Recorremos cada una de las filas obtenidas
while (rs.next()) {
    // Primer campo (la cero no existe)
    System.out.println( rs.getString(1) );

    // Columna "edad"
    System.out.println( rs.getInt("edad") );
}
```

Consideraciones sobre estos ejemplos:

- La sentencia es un **String** donde estará la consulta (Select, Insert, etc.)
- El método **executeQuery()** devuelve la cantidad de filas afectadas como un objeto ResultSet. Atención, este método sólo debe utilizarse en consultas que no modifican ni alteran la estructura de la base de datos; por tanto, executeQuery() debe utilizarse en consultas **SELECT**. Para el resto de consultas, utilizaremos **executeUpdate()**.
- El objeto **ResultSet** se compone de filas y columnas que representan los registros devueltos por la consulta. Cada fila del objeto ResultSet representa un registro de la tabla y cada columna representa un campo de ese registro. Se pueden acceder a los valores de cada campo utilizando los métodos getXXX(), donde "XXX" es el tipo de datos del campo (por ejemplo**, getString()** para cadenas de texto; **getDouble()** para los decimales; **getDate()** para las fechas; etc).
- El método **rs.next()** fila el cursor en la primera fila de resultados. Cada vez que hagamos otro rs.next() avanza el curso una fila más. Devuelve un booleano indicando si quedan más filas.

> **Nota**
>
> Los cursores es un concepto fundamental en base de datos. De forma muy resumida, y en este contexto, permiten recorrer los registros (filas) resultados de una consulta de manera iterativa. En este caso pueden ser parecidos a los índices de los arrays.

Finalmente, cerramos las conexiones creadas. Principalmente se debe cerrar la conexión a la base de datos (del objeto *Connection*), pero también es conveniente cerrar los objetos *ResultSet* y el *Statement* (o *PreparedStatement*), y en el orden inverso a la creación:

```
rs.close();
s.close();
conn.close();
```

Otro ejemplo, utilizando esta vez el método executeUpdate() para una consulta de creación de tabla:

```
…
Statement s = conn.createStatement();
String sql = "CREATE TABLE personas (nombre VARCHAR(50), edad INT)";
sentencia.executeUpdate(sql);
…
s.close();
conn.close();
```

Un ejemplo más añadiendo una fila a la tabla. Atención también que en este caso utilizamos PreparedStatement en vez de Statement:

```
…
String insertSql = "INSERT INTO persona (nombre, edad) VALUES ('Anakin', 30)";

PreparedStatement s = conn.prepareStatement(sql);
int filasInsertadas = sentencia.executeUpdate(sql);

if (filasInsertadas > 0) {
    System.out.println("Se ha insertado correctamente en la tabla persona");
…
s.close();
conn.close();
```

Nota

En general, es conveniente utilizar PreparedStatement en vez de Statement. Sin embargo, por cuestiones de rendimiento, la regla a seguir es utilizar Statement para consultas de tipo DDL (Create table, Alter table, etc) y para el resto de tipo DML utilizar PreparedStatement.

Tratamiento de excepciones

Si la base de datos no existe o la sentencia está mal construida, o se intenta insertar un valor duplicado en una clave primaria, las instrucciones anteriores pueden arrojar errores. Por lo tanto, es importante manejar estas excepciones utilizando bloques try-catch.

Las excepciones dadas mayormente son:

- **ClassNotFoundException**: lanzada por la instrucción Class.forName(…).
- **SQLException**: lanzada por el resto de las instrucciones, por ejemplo:
    - DriverManager.getConnection(…, "", "");
    - createStatement()
    - prepareStatement(…)
    - executeUpdate()
    - executeQuery()

- next()
- getXXX()
- close()

Recordemos de unidades anteriores que para asegurarnos de que, pase lo que pase, se cierra la conexión, conviene hacerlo en el **bloque finally**:

```
Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;

try {
     // instrucciones susceptibles a lanzar excepción
} catch (SQL Exception e) {
    System.out.println("Error en alguna operación sobre la base de datos");
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    System.out.println("No se ha encontrado el driver.");
    e.printStackTrace();
} finally {
    try {
        if (rs != null)
            rs.close(); // Cerramos ResultSet
        if (ps != null)
            ps.close(); // Cerramos PreparedStatement
        if (conn != null);
            conn.close(); // Cerramos Connection
    }catch (SQLException e) {
        System.out.println("Error al cerrar alguna conexión con la bd.");
        e.printStackTrace();
    }
}
```

Ejemplo 1: Test de conexión con la base de datos

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestConexionBD {
  public static void main(String[] args) {
    Connection conn = null;
    try {
      // Cargamos el driver
      Class.forName("com.mysql.cj.jdbc.Driver");

      // Definimos la url, el usuario y el password
      String url = "jdbc:mysql://localhost/pruebabd";
      String usuario = "root"; // por defecto en phpmyadmin
      String contrasenya = ""; // por defecto en phpmyadmin
```

```
      // Establecemos la conexión con la bd
      conn = DriverManager.getConnection(url, usuario, contrasenya);

      // A Completar: Mensaje de que la conexión se ha realizado con éxito.

    } catch (SQLException e) {
      System.out.println("No se ha realizado la conexión.");
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      System.out.println("No se ha encontrado el driver.");
      e.printStackTrace();
    } finally {
      // Cerramos conexiones
      try {
        if (conn != null) {
          conn.close();
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Ejercicio 11.1

Completa el ejemplo anterior para que muestre un mensaje si la conexión se ha realizado con éxito.

Ejemplo 2: Consulta a la base de datos

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConsultaPersonas {
    public static void main(String[] args) {
        // URL de conexión a la base de datos
        String url = "jdbc:mysql://localhost:3306/personas";

        // Nombre de usuario y contraseña de la base de datos
        String usuario = "root";
        String contrasenya = "";

        // Query SQL para obtener todos los datos de la tabla "personas"
        String consulta = "SELECT * FROM personas";
```

```
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;


        try {
            // Cargar el driver JDBC
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Conectar a la base de datos
            conn = DriverManager.getConnection(url, usuario, contrasenya);

            // Crear un objeto PreparedStatement para ejecutar la consulta SQL
            ps = conn.prepareStatement(consulta);

            // Ejecutar la consulta y obtener los resultados
            rs = ps.executeQuery();

            // Mostrar los datos de las personas accediendo a los campos
            while (rs.next()) {
                String dni = rs.getString("dni");
                String nombre = rs.getString("nombre");
                int edad = rs.getInt("edad");
                System.out.println(dni + "\t" + nombre + "\t" + edad);
            }

        } catch (ClassNotFoundException e) {
            // Manejar excepciones de carga de driver JDBC
            System.out.println("No se pudo cargar el driver JDBC");
        } catch (SQLException e) {
            // Manejar excepciones de SQL
            System.out.println("Error al conectarse a la base de datos:");
        } finally {
            // Cerrar las conexiones
            try {
                if (rs != null) {
                    rs.close();
                }
                if (ps != null) {
                    ps.close();
                }
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                // Manejar excepciones de SQL al cerrar los objetos
                System.out.println("Error al cerrar las conexiones");
            }
        }
    }
}
```

Recorrido del objeto ResultSet

Ahora tenemos claro que para recorrer los registros (filas) y obtener los campos de una tabla de la base de datos utilizaremos el objeto *ResultSet*. Vamos a repasar los métodos más destacados de este objeto:

- **next():** sitúa el cursor (índice) en la posición siguiente.
- **previous():** sitúa el curso en la posición anterior.
- **first():** sitúa el cursor en la primera posición.
- **last():** sitúa el cursor en la última posición.
- **relative(n):** se sitúa n posiciones más que la actual (o menos si n es negativo).
- **getRow():** devuelve el número de fila del *ResultSet*. La primera es la 1. Si no apunta a ninguna fila, devuelve un 0.
- **getXXX(col):** devuelven el valor del *ResultSet* de la columna col en la fila actual. La columna puede indicarse por un número o por el nombre de la columna. El XXX debe ser el tipo de datos del campo que estamos devolviendo.

Por ejemplo, si queremos saber cuántas filas tiene un ResultSet para después recorrerlo:

```
rs.last();
System.out.println(rs.getRow());
rs.first();
do {
      // Recorremos el ResultSet
} while (rs.next());
```

Ejercicio 11.2

Modificar el "Ejemplo 2: Consulta a la base de datos" para que acceda a los campos de la tabla con referencia numérica y recorra las filas con un do-while.

3.3. Consultas SQL parametrizadas

En programación, es común necesitar ejecutar varias veces una misma consulta de bases de datos, pero con diferentes valores de entrada. Para evitar tener que reescribir la consulta cada vez, se puede utilizar una **consulta parametrizada** que permita pasarle los valores de entrada como parámetros en el momento de su ejecución.

La consulta parametrizada se crea con un marcador de posición **?** en lugar de los valores específicos, lo que permite la reutilización de la misma consulta con diferentes valores. En el momento de la ejecución, se establecen los valores de los parámetros utilizando el método *ps.setString(pos,valor)*.

Este enfoque puede mejorar significativamente la velocidad de procesamiento ya que se evita tener que preparar la consulta cada vez que se quiera ejecutar con diferentes valores. En lugar de eso, se prepara una vez y se utiliza varias veces con diferentes valores de parámetros.

Un ejemplo de consulta parametrizada sería este fragmento de código:

```
String dni = "12345678A";
String nombre = "Anakin";
int edad = 30;

String consulta = "INSERT INTO personas VALUES (?, ?, ?)";

// Preparamos la consulta parametrizada
PreparedStatement ps;
ps = con.prepareStatement(consulta);

// Establecemos los valores de los parámetros utilizados
ps.setString(1, dni);
ps.setString(2, nombre);
ps.setInt(3, edad);

// Lanzamos la consulta
ps.executeUpdate();
```

Ejercicio 11.3

Modificar el ejemplo anterior para que realice tantas inserciones de personas como desee el usuario. El programa pedirá al usuario los datos de la persona, le mostrará si se ha insertado con éxito y le preguntará si desea continuar insertando personas en la base de datos.

3.4. Consultas DDL

Las consultas **DDL** (Data Definition Language) son aquellas que permiten definir y manipular la estructura de la base de datos. Las más destacadas son:

- **CREATE DATABASE:** crea una nueva base de datos. Ten en cuenta que si deseas crear una base de datos en tu programa, la conexión la haces sobre la url del servidor y no sobre una de sus bases de datos.
- **CREATE TABLE:** crea una nueva tabla en la base de datos.
- **ALTER TABLE:** modifica la estructura de una tabla existente, como insertar o eliminar una columna o cambiar el tipo de datos de una columna.
- **DROP TABLE o DATABASE:** elimina una tabla o base de datos existente en la base de datos o servidor.

> **Nota**
>
> Recuerda que debes usar el método executeUpdate() para este tipo de consultas.

Ejemplo 3: Creación de una BD

Fragmento de código para crear una base de datos en JDBC:

```
// URL de conexión a la base de datos
String url = "jdbc:mysql://localhost:3306/"; // No debes indicar la BD en la url

// Nombre de usuario y contraseña de la base de datos
String usuario = "root";
String contrasenya = "";

// Paso 2: Abrir la conexión
Connection conn = DriverManager.getConnection(url, usuario, contrasenya);

// Paso 3: Ejecutar una consulta SQL
System.out.println("Creando base de datos...");
stmt = conn.createStatement();

String sql = "CREATE DATABASE instituto";
stmt.executeUpdate(sql);
System.out.println("Base de datos creada exitosamente...");
```

> **Ejercicio 11.4**
>
> Completa el ejemplo anterior para que se ejecute como un programa de Java completo. Debes tratar las excepciones y cerrar las conexiones.

A partir del **ejercicio 11.4** realiza los siguientes ejercicios:

Nota

Para estos ejercicios necesitas conocimientos básicos de SQL. Si no has estudiado Bases de Datos puedes repasar estos conceptos básicos en el siguiente enlace:

https://www.w3schools.com/sql/sql_create_db.asp

Ejercicio 11.5

Escribe un programa que cree una tabla ALUMNOS que contenga los siguientes campos: nia (identificador único del alumno), nombre, edad.

Ejercicio 11.6

Escribe un programa que inserte tantos alumnos como pida el usuario.

Ejercicio 11.7

Escribe un programa que modifique la edad de un alumno/a de la base de datos.

Ejercicio 11.8

Escribe un programa que elimine una alumno/a de la base de datos. El programa pedirá el usuario el NIA del alumno/a y, si existe, lo eliminará.

4. Bibliografía

Basado en los apuntes de Espe Micó y Joan Gerard Camarena (IES Jaume II el Just, Tavernes de la Valldigna)

https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/